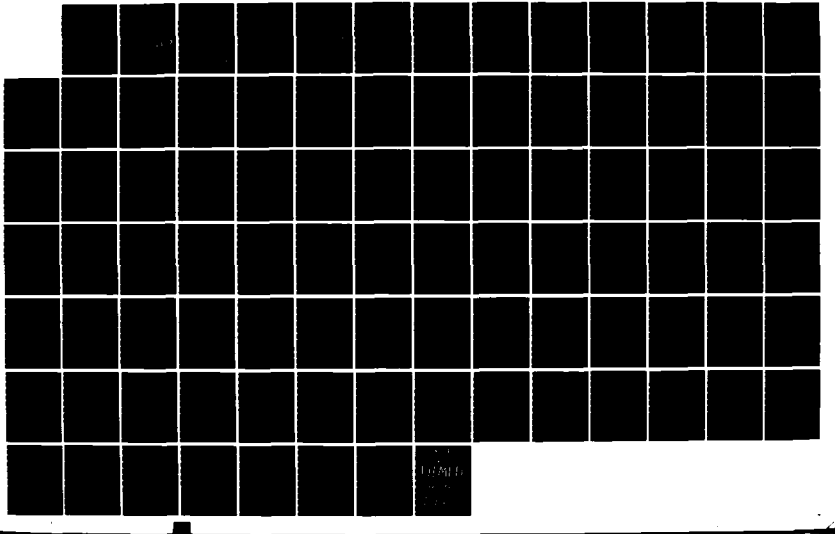
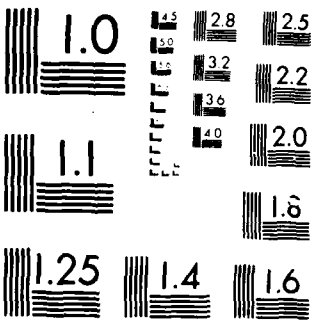
AD-A165 799 SILICON COMPILATION OF VERY HIGH LEVEL LANGUAGES(U) 1/1
ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE M W KAHRS
OCT 84 TR-145 N00014-78-C-0164

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART

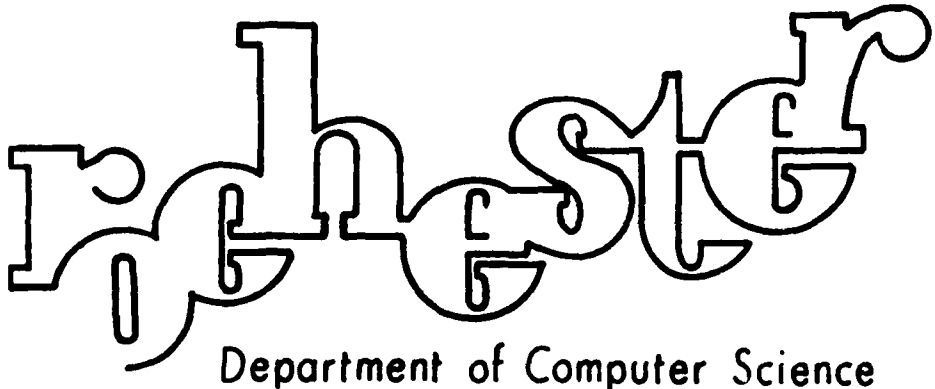
(6)

AD-A165 799

Silicon Compilation of Very High Level Languages

Mark W. Kahrs
Computer Science Department
The University of Rochester
Rochester, NY 14627

TR145
October, 1984



Department of Computer Science
University of Rochester
Rochester, New York 14627

Silicon Compilation of Very High Level Languages

Mark W. Kahrs
Computer Science Department
The University of Rochester
Rochester, NY 14627

TR145
October, 1984

Abstract

The report concerns the design and implementation of a compiler for two Very High Level Languages. The first language is a set language similar to VERS or SETL. The second language is a novel signal processing language. The compiler uses data flow and type information to constrain possible choices before choosing a possible implementation. Heuristic search is then used to choose from competing implementations of abstract data types. Constraint propagation is used at every selection step to remove incompatible configurations from the search. Finally, the use of specialized procedures called "design critics" is proposed to resolve global constraint conflicts. The output of the compiler is a parts list, a net list of module interconnections and the fields of the control store.

The preparation of this paper was supported in part by National Science Foundation Grants No. IST-8012418 and MCS-8104008, and in part by the Defense Advanced Research Projects Agency, monitored by the ONR, under Contract No. N00014-78-C-0164.

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1 REPORT NUMBER: TR 145	2. GOVT ACCESSION NO.: AD-A165799
	3. RECIPIENT'S CATALOG NUMBER:
4. TITLE *(and Subtitle)*: Silicon Compilation of Very High Level Languages	5. TYPE OF REPORT & PERIOD COVERED: technical report
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Mark W. Kahrs	8. CONTRACT OR GRANT NUMBER(s): N00014-78-C-0164.
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Science Department, University of Rochester, Rochester, New York 14627	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, Virginia 22209	12. REPORT DATE: October, 1984 13. NUMBER OF PAGES: 128
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Office of Naval Research, Information Systems, Arlington, Virginia 22217	15. SECURITY CLASS. *(of this report)*: Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

integrated circuits, circuit design, VLSI, very high level languages,

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The report concerns the design and implementation of a compiler for two Very High Level Languages. The first language is a set language similar to VERS or SETL. The second language is a novel signal processing language. The compiler uses data flow and type information to constrain possible choices before choosing a possible implementation. Heuristic search is then used to choose from competing concrete implementations of abstract data types. Constraint propagation is used at every selection step to remove incompatible

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

configurations from the search. Finally, the use of specialized procedures called "design critics" is proposed to resolve global constraint conflicts. The output of the compiler is a parts list, a net list of module interconnections and the fields of the control store.

The work reported in this thesis has shown that:

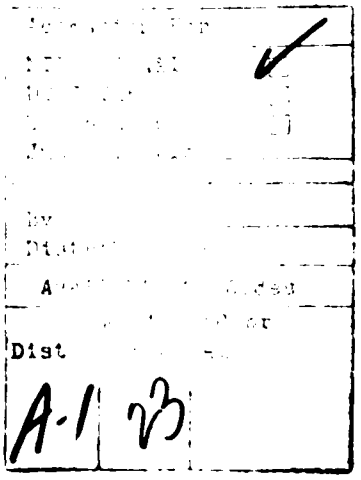
1) Compiler techniques can be used to generate machines from programs. These machines may then be implemented using VLSI modules.

2) Very High Level Languages can be used to hide the implementation complexity of VLSI design.

3) Constraint methods are profitably applicable to the VLSI problem domain.

4) Heuristic search and constraints can be successfully used to choose between implementations with differing costs.

5) Resource constraints can be used to control the optimization of the design by triggering specialized code.



Silicon Compilation of Very High Level Languages

by

Mark William Kahrs

Submitted in Partial Fulfillment
of the
Requirements for the Degree

Doctor of Philosophy

Supervised by James Allen

Department of Computer Science

University of Rochester

Rochester, New York

1984

ABSTRACT

The design of integrated circuits is a time consuming task. As the density of the circuits increases, so will the design problems. Several methods have been proposed for reducing the design complexity for VLSI. Some of these methods include the use of stick diagrams and compaction, primitive silicon compilation and the automatic generation of machines from low level descriptions. The work presented in this thesis is a step toward the ultimate goal of compilation of programs to silicon.

The thesis concerns the design and implementation of a compiler for two Very High Level Languages. The first language is a set language similar to VERS or SETL. The second language is a novel signal processing language. The compiler uses data flow and type information to constrain possible choices before choosing a possible implementation. Heuristic search is then used to choose from competing concrete implementations of abstract data types. Constraint propagation is used at every selection step to remove incompatible configurations from the search. Finally, the use of specialized procedures called "design critics" is proposed to resolve global constraint conflicts. The output of the compiler is a parts list, a net list of module interconnections and the fields of the control store.

The system described above has been implemented on a VAX-11 computer in a dialect of Lisp. It demonstrates that existing compiler methodology can be effectively combined with Artificial Intelligence search techniques to perform selection of VLSI modules for a very high level language. The work reported in this thesis has shown that:

- Compiler techniques can be used to generate machines from programs. These machines may then be implemented using VLSI modules.
- Very High Level Languages can be used to hide the implementation complexity of VLSI design
- Constraint methods are profitably applicable to the VLSI problem domain
- Heuristic search and constraints can be successfully used to choose between implementations with differing costs
- Resource constraints can be used to control the optimization of the design by triggering specialized code

Curriculum Vitae

Mark William Kahrs was born at 10:50 a.m. on the 25th of October, 1952 in the Clinica "Villa Margherita" in Roma, Italia. He is the son of an American architect and artist. Relocated with his family to San Francisco in 1955, he attended public schools in San Francisco, Mill Valley and Palo Alto until 1970. In 1970, he entered Revelle College at the University of California, San Diego. He worked his way through college by working summers as a system programmer at Tymshare, Incorporated in Cupertino, California. He majored in Applied Physics and Information Science (with a minor in Music) and graduated in 1974 with high honors and a good tan. In the summer of 1974 he worked at the Xerox Palo Alto Research Center (PARC) as a research intern. He entered the University of California, Berkeley in September of 1974, but left after a year. After knocking around for six months, he joined the Center for Computer Research in Music and Acoustics at Stanford University (CCRMA) as a research programmer. While working for CCRMA he completed his Masters thesis for Berkeley at the Xerox Palo Alto Research Center. After nearly two years at CCRMA, he moved on to Rochester. During his stay at Rochester he has worked on various occasions at the Institute pour Recherche et Coordination Acoustique Musique (IRCAM) in Paris, France. At Rochester he has been a teaching assistant and a research assistant. He has been responsible for various pieces of software and hardware. He has also co-authored a guide to Rochester for incoming graduate students as well as an introduction to the Computer Science Department. He has also organized numerous parties and Chinese banquets.

Acknowledgements

It's clearly impossible to try and thank everybody who encouraged me to finally get out, but I'm going to try anyway ...

It is customary to thank one's advisor first. James Allen has put up with me for an extraordinary time, even though the thesis doesn't have anything to do with Computational Linguistics. His readings of my drafts are responsible for any clarity present in the final version.

The rest of my committee, Jerry Feldman, Gershon Kedem and Dave Farden have all contributed to the final version you see before you. Jim Low offered me sage words at various times that helped me avoid some sticky tar pits.

During the past year, I have been supported by some of my best friends: Jon Austin, Diane Litman, Peter Selfridge and Ed Smith. They have enabled me to stay independent of an outside job and avoid distractions. Their generosity will not be easily forgotten.

My officemates of the past three years, Jim Heliotis and Lee Moore, have put up with my constant stream of paper pulp. Their patience with my pack-rat habits is appreciated.

Many other people in and out of the department have also helped by amusing, cajoling, distracting, dancing, eating, gossiping and sleazing with me. In particular, I wish to recognize the talents of Diane, Jill, Rose, Irene, Anni, Jon, Peter, Ed, Lee, Rick, Mayer, Frisch, Gary, James, Russell and Haas.

My lost friends on the west coast and my friends in Computer Music have given me good times and good cheer when I sorely needed both.

A long time ago, in a place far far away, Norm and Ann Hardy gave me my start in computer science. That start has given me more than I can easily state here.

A special mention goes to nano for lots of love, affection, food, wine and noodles.

Finally, my parents have put up with me being in school for over 25 years. Their love and support was critical to making it through those "grad school blues" and this wild eastern adventure.

This work was partially supported in part by NSF grants IST-8012418, MCS-8104008 and DARPA grant N00014-78-C-0164.

Table of Contents

Table of tables

Table of figures

"Depart not from the path which fate has assigned you"

Chapter 1

Introduction

1. Introduction

Designing any large scale digital system in any technology is a very hard and time consuming task. With semiconductor circuit density increasing, the design of larger and more complex systems and circuits will become nearly impossible. As an example, a modern 16 bit microprogrammed microprocessor such as the Motorola MC68000 took 100 man (person) months to design [FrS81]. This *excludes* the time required to layout the circuit.

2. Goals of the work

The overall goal of this work is the creation of a "silicon compiler". A silicon compiler is many things to many people. However, the idea expressed here is that a user unknowledgeable in the techniques of digital and VLSI design should be able to create a special purpose chip that runs the user's program. Specifically, the work reported here describes a system that compiles programs written in a Very High Level Language into a description of module interconnections. The modules are chosen from a library that was given to the system by the user. These modules have been designed using lower-level design tools.

To accomplish the goal of this research, the system described in the forthcoming chapters uses several techniques developed for use in Artificial Intelligence (commonly writ-

ten "A.I."). These techniques are necessary to overcome the complexity of designing large circuits like those found in today's microprocessors.

3. What this work reports

This thesis reports on the design and implementation of a VLSI compiler for a Very High Level Language. Data flow analysis is used to derive properties of variables. These "properties" are used to "preselect" the set of possible implementations. The compiler uses heuristic search to choose from competing implementations for an abstract data type. Type propagation is used to eliminate incompatible combinations, i.e., selections with conflicting input/output properties. Finally, if problems arise, then special design operators called "critics" could be called to try and resolve the design problems. (The implemented system did not have a full implementation of the critics). The final output from the compiler is a net list of module interconnections as well as a parts list and a listing of the microcode fields for the machine. The compiler assumes the existence of an automatic placement and routing system such as LTX [PDS77] or PI [Riv82]. The output from this subsystem should be suitable for direct implementation on silicon.

4. An example of system operation

As a demonstration of the capabilities of the system (called SILI, short for "Silicon"), consider the following program:

```
program transitiveClosure

set of set of integer : related, newlyRelated, found;
set of integer : x,y;
set with size 0 of integer : phi;

related := phi;
newlyRelated := base;
while (newlyRelated <> phi) do
begin
      found := phi;
      forall x in newlyRelated do
            forall y in x do
                  found := found with y;
      related := related with newlyRelated;
      newlyRelated := found - related
end

end.
```

Figure 1.1 Sample YASL program

This program is a slightly reworked example from Low's thesis [Low74] (pp. 14) written in the language "YASL" (Appendix C). Now assume the input library contains descriptions of sets implemented both as parallel and serial registers. A set library that contains these definitions can be found at the end of Appendix C. A full description of of the library format can be found in Appendix B. The full workings of the demonstration system are left for the subsequent chapters – in particular each chapter explains the functioning of one part of the SILI system.

The output of SILI is (1) a "parts" list (a list of modules) (2) a set of module interconnection graphs expressed as a net lists (3) a listing of the control store. Schematically, the output would look like the graph in figure 1.2.

One of the solutions given in Appendix C (YASL) is illustrated in figure 1.3. Note that the clock wires have been omitted.

SILI is both table driven and "language independent". As a demonstration of this, a very high level signal processing language called CLASP (discussed in Appendix D) was designed and implemented in SILI.

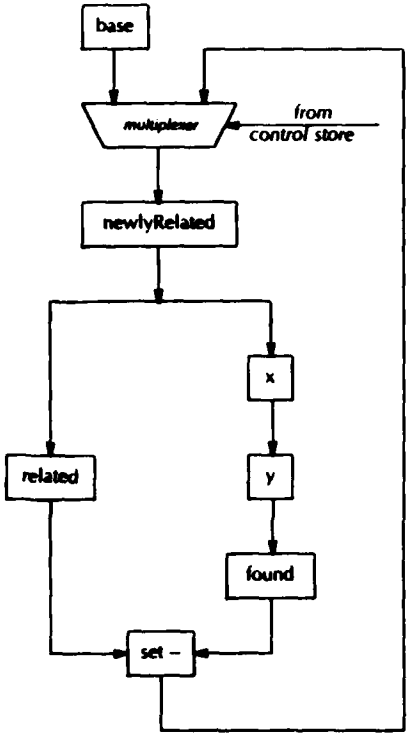


Figure 1.2 Data path section for the sample YASL program

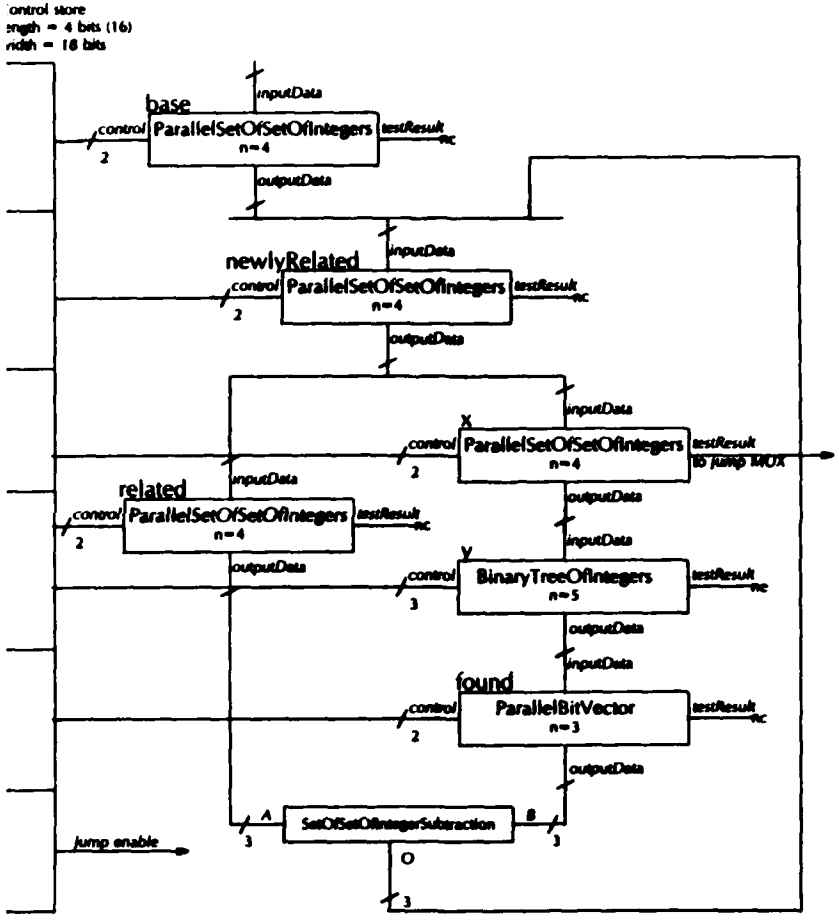


Figure 1.3 Schematic of the serial solution for the sample YASL program

The example shown below in figure 1.4 implements a well known touch-tone receiver:

```
module TouchToneDecoder

-- The now classic touch tone decoder, as done originally in 1963
-- by a group in Bell, then done again in 1968 by Jackson, et al
-- and done again by Lyon.

declare tuple of integer : lowerBand, upperBand;
declare tuple of integer : lowerBandCenterFrequencies;
declare tuple of integer : upperBandCenterFrequencies;
declare tuple of integer : detection;
declare integer : result; -- output from hum filter
                          -- (and iteration variable)
declare integer : bandLimit; -- bandpass (and lowpass) band limit
declare integer : input; -- input from the A/D
declare integer : output; -- output from the module
declare filter from 180 to INFINITY : noHum; -- Line hum filter

lowerBandCenterFrequencies := [ 697, 770, 852, 941 ] ;
upperBandCenterFrequencies := [ 1209, 1336, 1447, 1663 ] ;

result := noHum(input);
lowerBand := filter result from DC to 1070 : lowerGroupFilter;
upperBand := filter result from 1070 to INFINITY : upperGroupFilter;

detection := phi;

foreach centerFrequency in lowerBandCenterFrequencies do
      detection := detection plus
                   filter HalfWaveRectifier(
                       filter lowerBand
                         from centerFrequency-bandLimit
                         to centerFrequency+bandLimit
                             with Q of 15 and
                             with stopband attenuation of 16 db down
                             : lowerBandPass)
                       from DC
                       to centerFrequency+bandLimit
                   : detectLowGroup;

foreach centerFrequency in upperBandCenterFrequencies do
      detection := detection plus
                   filter HalfWaveRectifier(
                       filter upperBand
                         from centerFrequency-bandLimit
                         to centerFrequency+bandLimit
                             with Q of 15 and
                             with stopband attenuation of 16 db down
                             : upperBandPass)
                       from DC
                       to centerFrequency+bandLimit
                         : detectUpperGroup;

foreach result in detection do
      output := LevelDetect(result)

end.
```

Figure 1.4 Sample CLASP program

The detailed output from SILI for the sample YASL program (figure 1.1) can be found in Appendix C. The *simplified output (net list)* for one design can be found at the end of chapter 2. The output for the sample CLASP program (figure 1.4) can be found in Appendix D.

Note that throughout this thesis, references to the "sample program" refer to the sample YASL program, not the CLASP program.

5. Relevant work

The next three sections present an overview of work that is relevant to the thesis work. The first section looks at the present state of the art in design automation and claims the present design tools are not sufficiently powerful to solve the problems mentioned in the first section. The next section describes the use of A. I. based techniques in circuit design. The last section looks at the state of Very High Level Languages.

5.1. Design Automation

The computer aided design (CAD) of integrated circuits has been an active area of research for a long time. Therefore, the body of literature on CAD is long and extensive. Any attempt to comprehensively survey the field here would be inappropriate. However, the interested reader should see Newton's recent survey [New81] for a review of the current state-of-the-art in CAD for VLSI. The next two sections briefly review two aspects of CAD: graphic editors and layout languages.

5.1.1. Graphic Editors

Graphic editors were the first form of design automation to be used in the semiconductor industry. Rather than being laid out on paper, designs are laid out on a Cathode Ray Tube (CRT) using an editor which can eventually produce output suitable for fabrication (such as a mask description).

It has been observed that designs repeatedly use smaller designs. The latter designs are called "cells". Several graphic editors have been created that allow the user to define and call cells within the editor. Examples of these editors are ICARUS [FaR78], Daedalus [BMS81] and CAESAR [Ous81].

Like programmers, circuits designers control design complexity by exploiting hierarchy. This hierarchy can be expressed by a form of "macro expression" that occurs when cells are "called" within other cells. Self reference is not permitted as the graphic editors mentioned above lack the ability to stop recursion. However, as sophisticated as today's graphic editors are, the time to lay out an entire circuit is still overwhelming. In part, this is due to the limited size of the CRT screen. Only a certain number of devices can be displayed before the screen becomes a blur of color. The graphic editors mentioned above get around this problem by using "windowing" to present only a small part of the overall design. Unfortunately, while good for design in the small, windowing is terrible for design in the large. Another problem with graphic editors is the lack of signal typing. Without typing, connecting the output of one cell to the output of another cell (or connecting two signals that don't share a compatible format) is quite easy.

5.1.2. Layout languages

Work has also been done in using non-graphical languages to describe circuit layout. DPL [Bat81][BaH80] is a simple Lisp based command language that has low level primitives that can be used to form complex commands. For example, "(from (pt x1 y1)) (run-layer 'poly) (run-width 2) (tox deltaX) (toy deltaY)" will create a polysilicon line of width 2 lambda that runs from (x1, y1) to ((x1+deltaX), (y1+deltaY)). These primitives are used by Lisp functions written by the user to generate cells. Besides rudimentary layout functions DPL also provides primitives for iteration and PLA construction. One purpose of DPL is to provide flexibility at the cost of sophistication. In fact, DPL is really a meta-language to be used in constructing new design systems.

Shrobe's Data Path Generator (DPG) [Shr82] an example of a system built using DPL. It is used to construct the data path sections of special purpose machines. The cells used by the DPG are designed with the graphic editor Daedalus. The cells are also described with a declarative language for input to the DPG. The data path section generated by the DPG has a fixed architecture with drivers at the "top" and a global bus that connects the registers and operators together. The registers and operators have a fixed vertical pitch but are stretched at predefined stretch points to attached to other fixed wires. Shrobe points out that the DPG is organized around a particular design style that makes it inflexible when the problem domain is changed.

5.2. Silicon compilers

As mentioned in the introduction, the phrase "silicon compiler" has come to mean many different things – from compiling programmable logic arrays (PLAs) to generation of machines from languages. The next sections look at some systems that have been called "silicon compilers".

5.2.1. Bristle Blocks and Siclops

Bristle Blocks [Joh79] is a primitive silicon compiler. It accepts a specific microcode instruction word and a list of elements in the data path (called a "core" by Johannsen) and generates a two bus machine that implements the microprogram. There are three input sections to Bristle Blocks: the detailed microcode specification, the list of buses and bus

connections and the elements of the "core" (i.e., the cells). The library cells of Bristle Blocks have a fixed layout and are described procedurally, like DPL cells.

Siclops [HSC82] is a new implementation of Bristle Blocks. It was designed to overcome some of the limitations of Bristle Blocks. In particular, it allows a flexible floorplan as well as automatic routing of signals, power and ground.

5.2.2. MacPitts

MacPitts [SSC82] accepts a register transfer language with parallel constructs and generates a machine down to the level of layout. However, the architecture of the underlying machine seeps into the language specification. For example, all registers are read before they are written, which allows for some seemingly confusing constructions. For example, (par (setq a b) (setq b a)) interchanges a and b because both a and b are read out before they are written. Since registers are read before being written, this assures that the registers do not share buses. MacPitts guarantees this by copying data paths until there are no conflicts. Operations that are to be performed in parallel must be marked explicitly by using the "par" constructor as shown above. MacPitts also uses a primitive control transfer operations (goto) to effect state transfer. The cells used by MacPitts (called "organelles" by MacPitts designers) are defined using Lisp functions to be bit slices. The "organelles" generated by these functions must be able to stretch (like Shrobe's DPG cells) to connect to the signal and power buses.

5.2.3. CMU Design Automation System

The CMU Design Automation System [PTS79] [DPS] is a long term project with the eventual goal of automating digital design. The CMU DA System accepts a machine description written in *ISP* [Bar78] [Bar81], a particularly low level register transfer language. Although ISP is both flexible and general, its level of description is very low. If the designer does not have an architecture in mind, the system will not create an architectu

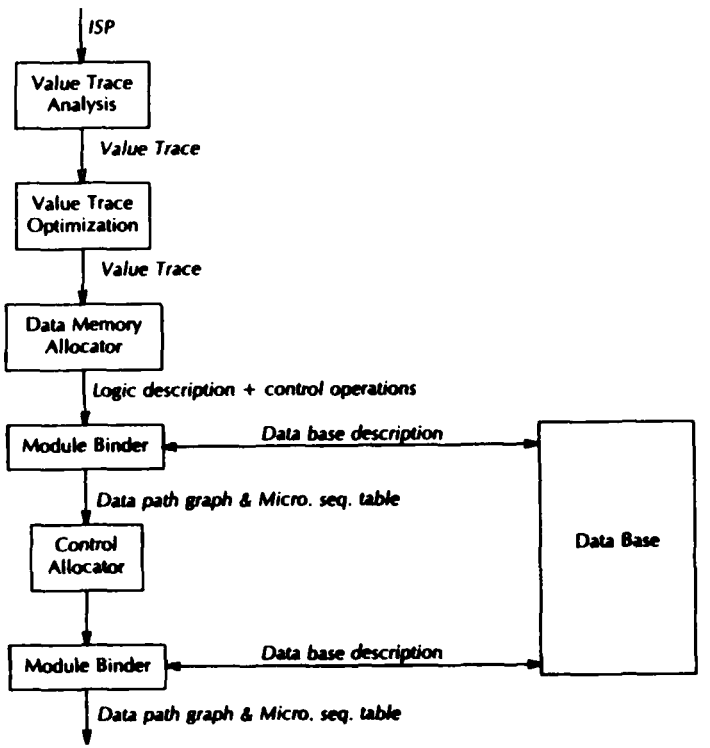


Figure 1.4 CMU DA system block diagram

First, the ISP description is parsed. The Value Trace analyzer runs over the parse tree generating the Value Trace graph. The Value Trace graph can be optimized, much like a program to get rid of inefficient operations. Next, a decision is made "*by the user*" about the "design style" to be used. A "design style" is a class of machine organizations, such as pipe-

lined, decentralized arithmetic and so forth. The design style allocator creates data paths from the Value Trace. The module binder assigns implementations from the library to the various modules. The final step is the generation of the control section of the machine.

The CMU DA system differs from SALI in many ways. First, the input language of CMU system is ISP, a low level register to register transfer language. ISP comes close to dictating the internal architecture of the machine completely by specifying the connection of internal registers.

The value trace of the CMU system differs in a significant way from the control and data flow analysis used in SALI: variables are removed from the ISP, whereas in SALI, the variables are used to select implementations.

The optimization phase of the CMU system is similar to the use of critics in SALI. However, some optimizations of the CMU system (such as the transformation of parallel to serial designs) are not needed here since the selection phase will choose these designs automatically.

The module binder phase of the CMU system has the same goal as the selection phase of SALI, i.e., attaching specific matches to the data path of the machine.

Likewise, the control allocator is very similar to the control section synthesizer of SALI except that the control section synthesizer operates from a control flow graph and not from a "procedural description" as in the CMU DA system. The control section synthesizer reflects the "microcode style" of the CMU DA systems.

5.2.4. MIMOLA

The MIMOLA design system [Zim79][Zim80] is a system with similar goals to SALI, but at a much lower level. The MIMOLA system accepts a macro language with low level constructs and generates a register transfer machine that implements the program. The next step in synthesis (the Automated Logic Design System) takes the register transfer machine and

generates a gate level description by doing macro expansion and tree walking. The last step in the design process is the use of the Physical Design System (a subsystem of MIMOLA) to complete the task of layout. More recently, the MIMOLA system has been focused on the VLSI domain. The system is now called the "MIMOLA Software System" (MSS) [Zim81] The MSS user can put specific limits on resource usage. MSS uses constraints to control resource allocation (See Chapter 3). Perhaps the biggest difference between MSS and SALI is the view of the user. MSS is a "full partner" in the design process; in SALI, the designer supplies a program and a few assertions and the rest is automatic.

5.2.5. ARSENIC and Xi

The LISA language (part of the ARSENIC system [Gaj82]) and Xi [Joh] are similar systems. They are both high level languages with similar approaches to compilation. Both languages have fairly primitive operators (such as "shift" and "rotate") that are expressed directly in the generated hardware. ARSENIC is a top down design system of which LISA is only one part. Like SALI, it performs timing analysis on the generated hardware to check that the chip meets the design constraints.

5.3. A.I. approaches to automating circuit design

Some work has been done using A.I. principles and techniques in automating construction of circuits. McDermott's [McD77] thesis system, "DESI", was able to design simple circuits from a plan. McDermott's system used constraints between modules to guarantee that the modules would work together. A similar technique was used by a group at Rutgers [MSS81] [KeS82] to analyze behavior of digital circuits. deKleer's thesis proposal [deK] worked on a theory of planning that accounted for some of the deficiencies of McDermott's thesis. Specifically, these deficiencies included the lack of knowledge of partial plans and the ability to recognize circuit fragments as plans. Brown's thesis [Bro76] is closely related to McDermott's but deals with the problems of debugging circuits, not generation.

These works will be discussed further in the chapter on constraints. However, the planning view of design is quite different from the compiling approach used in SILI. In planning, the user provides the system with a goal and constructs plans to achieve that goal. A hierarchical planner divides the main goal into subgoals (recursively) until the subgoals are achievable. (See Nilsson [Nil80] for a review of planning). Generating a plan from a high level goal is an extremely difficult task. First, the planner should have a data base of available plans. The plans must have preconditions, effects (and possibly constraints). Second, the planning mechanism should be able to carry many plans in parallel. In compiling, the user provides a program (a specific solution to a problem), not a plan, and this program is used to generate the output code that will produce the solution. SILI outputs a description of the machine, not code (except for the microcode of the machine!).

Compilers accept *programs* i.e., a specific solution to a problem. A planner accepts a high level *goal*.

5.4. Very High Level Languages

In 1973, Earley [Ear73] identified three criteria for the design of (very) high level programming languages. They were

(1) The ability to write a program in a clear and concise manner
(2) The ability to ignore the implementation issues and concentrate on the semantics and correctness of the algorithm
(3) Postpone design decisions on seemingly unrelated portions of the program until needed.

Of these three points, the second point is of particular interest. This is because the user of a high level silicon compiler *should* be able to ignore the details of VLSI implementation.

Past work in Very High Level Languages has been principally done in languages with abstract types such as sets, tuples and relations. As stated above, the use of these high level types intentionally obscures common programming details such as pointer chasing, memory allocation, structure formation and implementation selection.

SETL [Sch5)] [DeS79][Sch75] is perhaps the best known set language. SETL uses sets as its fundamental data type. It has existential and universal quantifiers as primitive operators as well as functions over sets. Since its inception, SETL has been studied as a vehicle to explore automatic selection.

VERS2 [Ear73] [Ear74] was another Very High Level Language developed by Jay Earley and students. VERS2 used the notion of "relational access" rather than the notion of "access paths" (e.g., pointer references) for data structure access in programs. It shared many features with SETL but had a different syntax and a relational and matching sublanguage.

SILI uses two very high level languages; one, called "YASL" is a descendant of VERS2. It is discussed in Appendix C. The other language is a signal processing language called "CLASP". It is discussed further in Appendix D.

6. Organization of the thesis

This thesis begins with the example program shown earlier in figure 1.1. This example is used throughout the thesis to demonstrate the parts of the system. The second chapter is an overview of the system, what each part does, what it connects to and how the symbiotic whole works. The third chapter describes the use of constraint (type) propagation in reducing the search space. The next chapter introduces heuristic search as a method for choosing implementations, and includes descriptions of the metrics and module binding techniques. Chapter 5 presents the implicit machine model and how to construct microcode for it. Chapter 6 discusses the implicit machine architecture as well as how to generate the control store. Chapter 7 introduces the use of critics as a mechanism to resolve constraint failure. Each possible critic is also described along with the circumstances that could bring about its use. The eighth and final chapter presents the results of the work: what worked and what didn't as well as the organization of SILI. Appendices are included that give extremely

detailed annotated traces of SILI in operation.

"You will take a chance on something in the near future"

Chapter 2

Overview of SILI

1. Introduction

In this chapter, SILI is surveyed. The chapter is self standing, i.e., can be read without reading the remaining parts of the thesis. The previous chapter introduced the problem of VLSI compilation and past work. The reader is encouraged to read it for further background on the problem.

Section 2.1 briefly discusses the various components of SILI with regard to their input/output behavior. The detailed descriptions of the innermost workings of each component are described in the subsequent chapters.

The input to the system is discussed in section 2.2. SILI was designed to be "language independent", i.e., the internals of SILI are *not* dependent on the semantics of the input language. The exact syntax and semantics of the input languages that were used are left to Appendices C and D. Next, the concerns that led to the inclusion of particular information are discussed. The next section discusses how heuristic search and type matching work together as a selection mechanism. After finding selections, the next step is the generation of the control store and the control machine. Global constraints and critics are discussed next as a mechanism of design modification in the event of selection failure. Finally, the part of SILI that actually generates the output is presented.

2. The modules and their input/output behavior

The overall organization of the compiler is shown below in the figure 2.1:

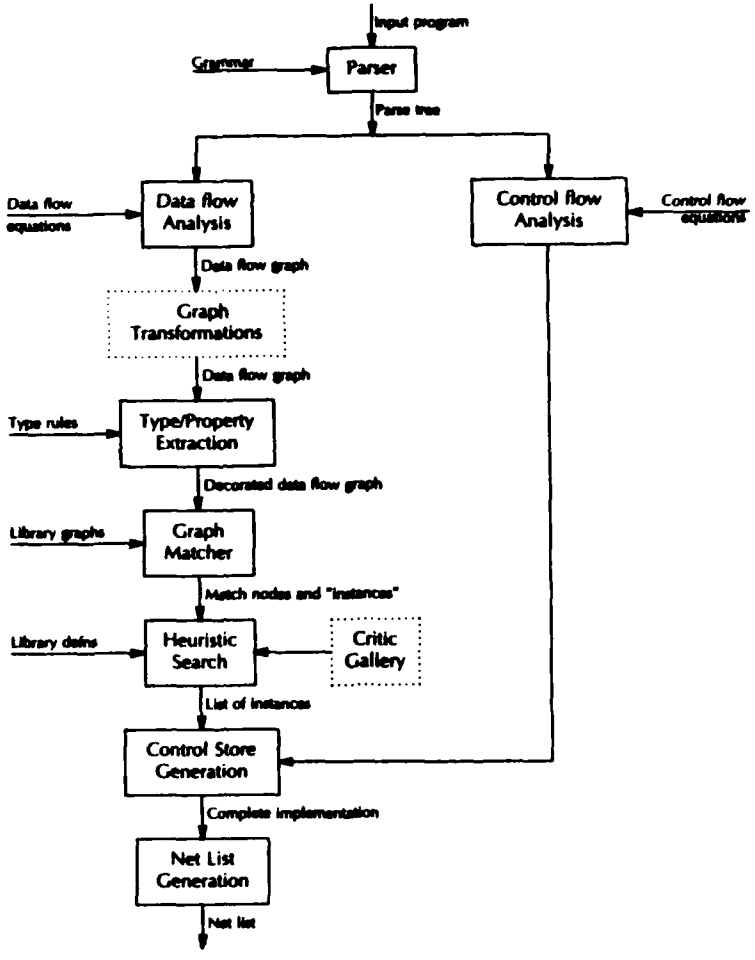


Figure 2.1 Overall system organization

The output from the parser is a parse tree. It is used by several modules including the type declaration module (not shown in the figure) and the flow analysis modules. The type declaration module just "decorates" the tree with various type declarations from the declaration statements. The control flow procedure now takes the parse tree and calls upon the data flow procedure to analyze subtrees of the parse tree at appropriate points. The output of the control flow and data flow analysis routines are the control flow and data flow graphs.

It is now possible to apply various graph transformations (for example, algebraic transformations) at this stage. Cattell's thesis [Cat78] includes a catalog of various possible transformations (see page 52) that would be useful in a system that is in production use. However, the implemented Slu system did not perform any transformations on the data flow graph. (Note that unimplemented sections are illustrated using dotted boxed).

The next step is to match the data path (as described by the data flow graph) against the various implementations described in the library. The matching procedure uses the type information to cut down the number of possible implementations during the search. The matcher establishes a correspondence between the matches (called "instances") and the nodes in the data flow graph.

Having built up the correspondence between instances and nodes in the data flow graph, the next step is to "bind" the parameters in the library description to the properties in the data flow node. This "fully instantiates" the node by specifying parameters like size and width. The next step is to choose the implementations.

The selection of implementations (instances) begins by sorting the nodes of the data flow graph by the number of implementations that matched that node. This is because the nodes with fewer choices should be chosen first as they will constrain the choices later on. After sorting the implementations, search begins. As the search is taking place, each choice is "weighed" to guarantee that the choices will be within the design constraints specified by

the system user. Should a choice violate one of these constraints, then a design "critic" would be called in an attempt to resolve the conflict. (The implemented SILI system did not have full implementation of critics). If resolution is not possible, the choice is tossed out. Also, as each choice is made, the signal types of the implementations are checked to make sure that they agree. If they don't, then the choice is thrown out. The search process ends when all nodes have been processed. The next stage generates the control section.

The generation of the control section is relatively straight-forward. It begins by gathering all the control fields of the nodes in the control flow graph. Next, a control section is created from the library and the fields of the control section are filled. The final step is the generation of a multiplexer that gives the jump signal to the control unit. (The multiplexer serves to gate the appropriate test signal to the program counter).

The final step in the compiler is to generate the net list. This is done by connecting each module, one at a time, to the modules which its signals are connected to.

The rest of this chapter is devoted to taking the example from the previous chapter and "running" it through the system.

3. Information gathering

A "conventional" compiler, i.e., one that compiles a source text into a machine language needs to know about data types, data flow and control flow as well as various facts about the output machine language and machine model.

SILI requires more information than a conventional compiler. Besides the abovementioned data, a VLSI compiler needs information on the library of implementation choices as well as the implementation constraints of each choice.

In the next three sections, the front end of the compiler is examined to show what types of information are needed and gathered.

3.1. Data flow analysis

Data flow analysis is an old idea dating back to the earlier days of compilers. Recently, data flow analysis has benefited from a rigorous development which can be found in Hecht [Hec77] or Kennedy [Ken81]. Data flow analysis derives a directed graph called (appropriately) a data flow graph. Each node in the graph corresponds to a variable or operator in the input program. An arc connects two nodes if data can be "transmitted" from the source node to the sink node. For example, take the example program shown in figure 1.1. Its data flow graph is shown below.

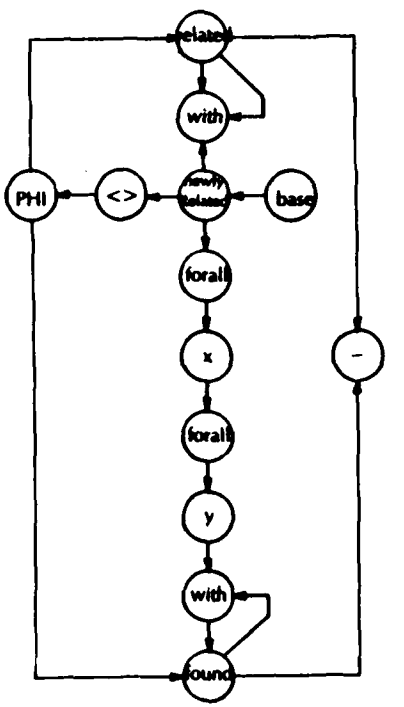


Figure 2.2 Data flow graph for example program

Fortunately for system designers and implementors, data flow graphs are easily constructed at parse time. Rosen [Ros77] and Kennedy [Ken81] give procedures that can be executed as semantic routines during parsing, i.e., as a form of syntax directed translation.

The data flow techniques used by SIL will not be described here because they are not critical to the operation of the system. The data flow technique is a bit unusual because the system is "language independent" and therefore the analysis is table driven. The interested reader can find all the details of the method in Appendix A.

3.2. Control flow analysis

Control flow analysis is closely related to data flow analysis. It describes the path of the program counter as the program is executed. Like the data flow graph, it is a directed graph where each node represents a statement and each arc represents the transfer of control from the source to the sink.

Like data flow graphs, control flow graphs are easily formed at parse time. Like the data flow method, the control flow analysis procedure used by SIL is language independent and table driven. The details of the method can also be found in Appendix A.

Allen [All70] has an introduction to control flow analysis. A more modern introduction can be found in Aho and Ullman [AhU77].

For the example program, the control flow graph is shown on the next page.

3.3. Other forms of analysis

There are other analytical techniques that are of use in gathering information about a program. For example, the range of an array or the maximum size of a set will be extremely useful during the selection process because these measurements are used to generate the appropriate sized elements. The use of this information will be discussed in greater detail in the section on binding (see Chapter 4).

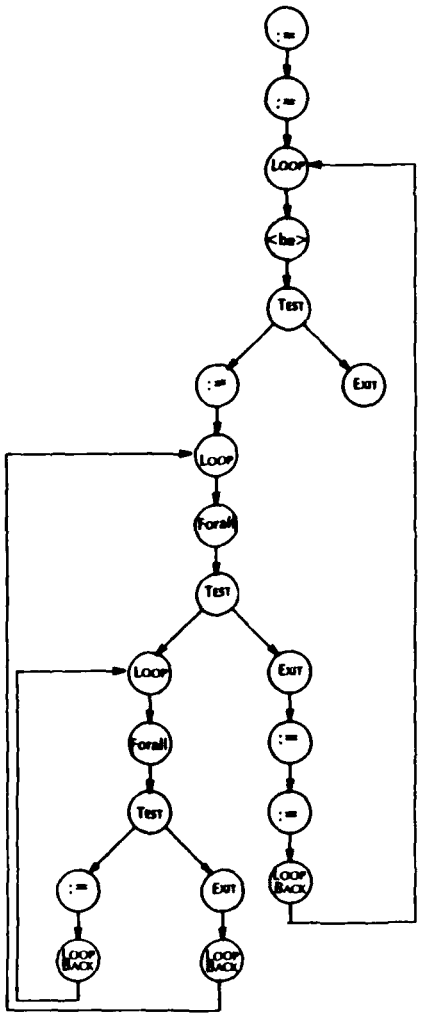


Figure 2.3 Control flow graph of sample program

3.3.1. Property Extraction

Besides control and data flow analysis, property extraction is another useful compile-time technique.

3.3.1.1. Type determination

Tenenbaum [Ten74] and Kaplan and Ullman [KaU80] both present algorithms for determining type in languages with runtime typing. Wegbreit [Weg75] presents an algorithm that computes various program "properties" including the type of variables (in a language with *runtime* types) and data bounds. Type information is of immediate use to a VLSI compiler that separates the implementations by type as this compiler does. Type propagation is a needed component in the system because the interior nodes of the data flow graph (the operators) do not have types declared explicitly. A modification of Wegbreit's procedure could be used to derive the types of the interior nodes in the data flow graph. This will become clearer in the section on binding in Chapter 3.

3.3.1.2. Other properties

Suzuki and Ishihata [Sui77] constructed a special purpose theorem prover to check array bounds of Pascal-like programs. A modification of their technique could be useful in establishing the size of arrays or sets at compile time.

4. Machine Generation

Every compiler has a machine model. Most often, this model is embedded in the code generator. For example, the code generator for the VAX [Str78] should generate code that takes advantage of the machine's multiple registers, varying data formats and addressing modes. The compiler discussed in this thesis must take a slightly different tack; it is generating machines, not instructions. Therefore, the notion of generating code for a machine becomes one of generating a machine for an algorithm.

The step taken in this thesis is to generate a so called "Harvard Machine" for the input algorithm by transforming the data flow graph into a multi-register machine and the control flow graph into a control section that controls the register machine.

As an example, take the algorithm presented in section 1 of this chapter. The derived dataflow graph for this was shown in figure 2.2 above. Now, if each variable (node) in the graph becomes a register or operator and each arc becomes a data path, then the graph becomes a simple machine as was shown earlier.

The next phase is the selection of the implementations for the variables and the operators. A "conventional" compiler has a machine model and a fixed set of resources. The problem there is to generate code for the machine and use as little of the available resources as possible. The problem faced here is to generate a machine that correctly and efficiently implements the input program.

An architecture is an implementation of a computational model. So, after choosing a model, the problem becomes one of generating an architecture. This will be discussed in the context of implementation selection.

At first glance such a machine would be unrealizably large. The purpose of the design critics discussed in Chapter 7 is to compress the machine by changing the underlying architecture. For example, one change might be to bus several data paths together.

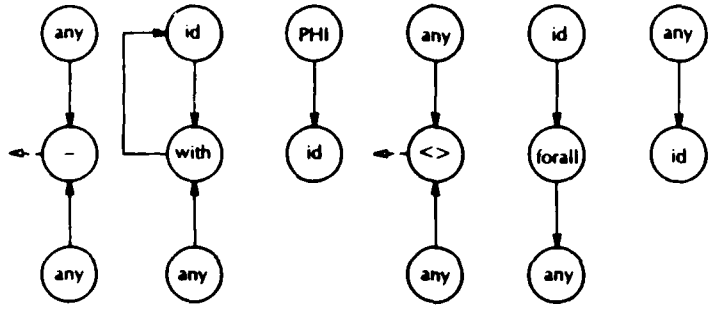
5. Implementation selection

5.1. Matching

Each library implementation description has a set of data flow subgraphs. These represent the data flow graph computed by the module for each combination of control signals. Matching must be done between the data flow subgraphs of the library implementations and the data flow graph of the input program. The matching technique will be described in

detail in Chapter 4. The result of the matching procedure is a pairing between the nodes in the data flow graph of the program and the nodes in the description of the implementation. Consider the current example. In order to match the data flow graph of the program (shown earlier in figure 2.2), the subgraphs shown in the figure below must be described in the library.

Data flow subgraphs



Match number

Figure 2.4 Required data flow subgraphs for sample program

After matching the subgraphs of the library against the graph of the input program, the match that results is shown on the next page in figure 2.5.

Note that the matching procedure matches both nodes and types that are "bound" to the nodes by type analysis. This has the effect of partitioning the library by type.

After matching program nodes with implementation nodes, the next step is to choose the appropriate matches. This is commonly called "implementation selection".

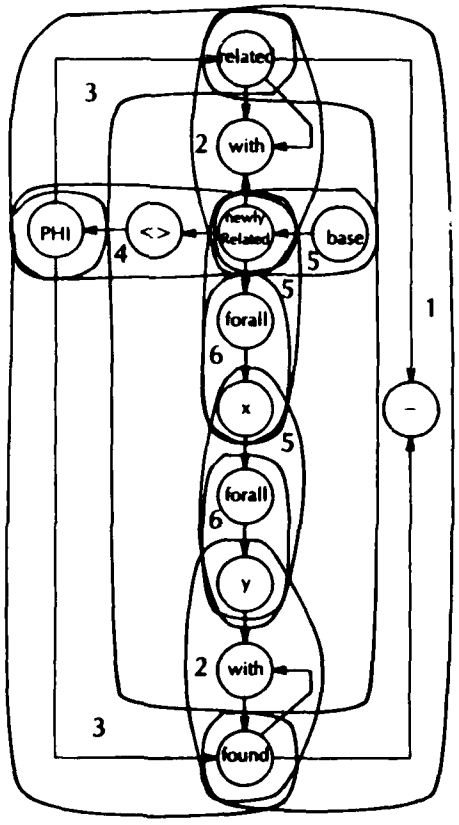


Figure 2.5 Matched data flow graph of sample program

5.2. Selection

The problem of selection is one of choosing an implementation from a list of possible *implementations*. In SEL, a selection chooses a concrete implementation of an abstract data type. For example, a hash table is a concrete implementation of a set. Of course, each selection carries with it various costs, including time, area and power. The selection process

described in chapter 5 uses these parameters to guide the implementation search.

Previous work in implementation selection has centered around the use of three techniques: heuristic search [Low74][Rov], selection rules [Kan82] and program transformation [Pai83]. In heuristic search (the approach taken by this thesis), the selection is done using a tree search with a heuristic evaluation function. The selection rule systems use production rules to do selection based on the preconditions of the rules. The program transformation technique continually transforms the input program until the program can be compiled using the implementations in the library.

The search procedure takes as input a library of possible implementations, a machine generated from the data flow graph, a type interaction table and a list of global constraints. A heuristic search technique is used to guide the selection process.

The first step is the act of binding. Binding is the step where program properties are used to derive implementation properties. For example, set size becomes the number of bits in a bit vector implementation of a set and so forth. Binding is accomplished through the use of interpretation routines that are specified as part of the library.

Next, as each selection is made, input/output types are checked between the input and output ports of the chosen implementation. If it is impossible for the signals to match, the selection will be thrown out. This process continues until either all the nodes in the machine have implementations or no selection is possible. In the latter case, a design procedure, called a "critic", could be called in an attempt to resolve the conflict.

As an example, consider the possible implementations for the variables and operators in the data flow graph of figure 2.2. Figure 2.6 on the next page illustrates the "schematics" that are available for set implementations in the YASL library.

It is possible for the selection algorithm to terminate with more than one choice and it is also possible that none of the selections will be optimal for the problem.

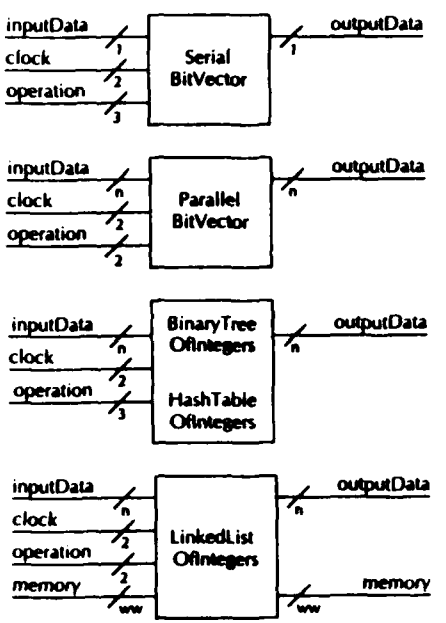


Figure 2.6 Sample library modules

6. Generation

After selecting implementations for the nodes in the machine graph, the next step is the generation of the control section of the machine. It will be shown later that it is necessary to delay control section generation until after the selection of implementations has been made. Since the generated machine is assumed to be synchronous, the control of the machine is

made by a microcoded store. The process of control store generation begins with analysis of the control flow graph and the data flow graph bound with implementations. The process of generating the microprogram proceeds by examining each node of the data flow graph for the control field of the implementation module. The fields are collected to form the microprogram word.

Next, the control flow graph is walked from head to tail generating assignments to each field in the microprogram word. The last step in generation is to create the control store and generate a multiplexer for the jump signals. The control section for the sample program (for the all parallel implementation) is as follows:

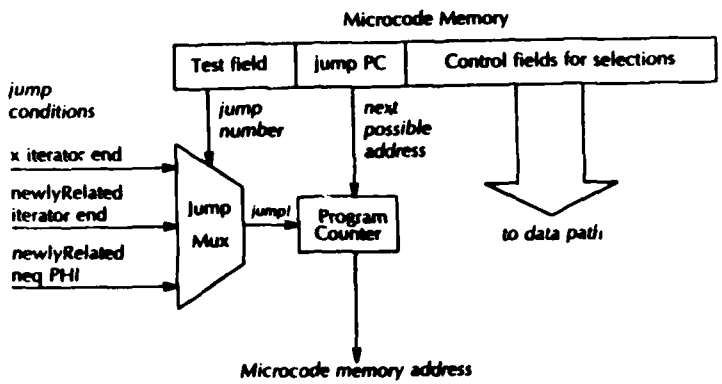


Figure 2.7 Control section for sample program

The final step is the conversion of the machine graph to a "net list" (a connection list) suitable for a placement and routing system. The net list of the example program for the binary tree version is shown below:


```
SetOfSetOfIntegerSubtraction: outputData/3 to ParallelSetOfSetOfInteger: inputData/4
ParallelBitVector: outputData/3 to ParallelSetOfSetOfInteger: inputData/4
ParallelSetOfSetOfInteger: outputData/4 to SetOfSetOfIntegerSubtraction: inputA/3
ParallelSetOfSetOfInteger: outputData/4 to ParallelSetOfSetOfInteger: inputData/4
ParallelSetOfSetOfInteger: outputData/4 to ParallelSetOfSetOfInteger: inputData/4
ParallelSetOfSetOfInteger: outputData/4 to SetOfSetOfIntegerSubtraction: inputB/3
```

If you wish to, you will have an opportunity"

Chapter 3

Constraints

1. Introduction

The word "constraints" is now a catchword for several different problem solving methods in Artificial Intelligence. These techniques will be discussed further in the next three sections. The first section introduces constraints and how they can be used in problem solving systems. The next section covers the use of constraints in a silicon compiler. The last section discusses relevant past work in constraint based systems.

2. Introduction to constraints and problem solving

2.1. Representation

A constraint can be broadly defined as a restriction that specifies the range of values of variables in an equation. It is often easiest to express these restrictions as equations. As an example, consider the analog circuit domain of Stallman and Sussman [SuS75]. A constraint in their system might take the form of an electrical equation involving other nodes in the circuit being analyzed. In the simplest form, however, constraints can be represented as relations. An example of this is Waltz' thesis system [Wal75], where the constraints are restrictions on the labelings of arcs. Because of the differing complexities of the representations, different problem solving techniques are used to find solutions to the constraints. These

techniques will be covered in the section on relevant work.

2.2. Use in problem solving

Given a constraint, the next question is: how does SILI compute the values of the variables in the constraint equation (or relation). Stallman and Sussman use a combination of symbolic algebraic manipulation and "propagation" to solve the circuit equations. Here, "propagation" refers to using rules to assign values to the nodes in the circuit. When the system uses relations, the propagation step is easier. A search technique is used to choose possible values for the variables. Then, the applicable constraint relations are evaluated. If the constraints are satisfied, then the variables are instantiated, otherwise, the variables are "thrown out".

3. Use of constraints in VLSI design

3.1. Introduction

Constraints are present at all levels of VLSI design. At the bottom level, constraints called design rules specify the minimum spacing of lines. The next level up is the transistor level; e.g. transistor t_1 must have a ratio of 2:1 with transistor t_2. The next level up are cells. Typical inter-cell constraints are of the form "port q' doesn't have superbuffers, therefore it must be close to its sink". Cells make up modules and module constraints specify properties. For example, "module M, port inputA takes parallel, two's complement integers". At the top level, modules are connected together to form systems. The constraints at the system level are global performance constraints.

In SILI, the lowest level of representation is the module layer. The inter-module constraints have a local nature; they exist between ports of the modules that are connected by data paths. These will be called "port constraints".

There are also more global constraints. These constraints express the high level performance and resource bounds. These will be called "specification constraints".

Finally, there are constraints that are specific to modules being matched. If these constraints are satisfied during matching, then the module is matched, otherwise the module is not matched. These constraints will be called "matching constraints".

3.2. Port constraints

There are two ideas behind the use of port constraints. The first purpose of the port constraints is to ensure that other modules connected to the port will be able to "talk" to the module (i.e., the ports share common typing). Note that every module in the library has semantics (or properties) associated with each port. If these semantics are matched, then the connection is valid.

The second purpose of port constraints is to establish data paths between modules. This is because when a module description is given to the system, there are no indications about which ports of which modules can connect to a particular port. Therefore, one method of connecting ports of modules is to match the types of the ports.

Specifically, consider the use of program properties in selection. If port constraints are defined as binary relations over types, then the constraints can be expressed as the relation $=(t_1, t_2)$, where t_1 and t_2 are the types of ports. The actual type matching takes place during selection. Each type of the ports on the newly selected module are traced backward to the connecting module. If there isn't a selection yet, then the port is deemed acceptable; the type checking will take place when the other module is selected. If the other module has been selected, then the type of the port on the other module is compared (using the relation "type-compare") with the type of the port on the newly selected module. If they match, then the selection is permitted to proceed. Otherwise, the selection is put into the "reject bin".

For example, take the case of two nodes n_1 and n_2 each with a parallel and serial implementation. Assume that n_1 and n_2 are connected, i.e., there is a data path between them. During the search phase of selection if n_1 is selected first then attempts to propagate the port constraints will fail as n_2 is not instantiated yet. When the search reaches n_2, then the port types of n_1 and n_2 are checked since they will both be instantiated.

Note that matching also involves searching – each port of every selection must be matched against every other port of the connected selections. For example, suppose port A has types t_1 and t_2 and port B has types t_3 and t_4. Then, type t_1 is checked with type t_3 and then type t_4. Likewise, this will happen with the type t_2. The demonstration program implements this by using a depth-first search.

3.2.1. An example

Consider the following subgraph of the data flow graph shown in figure 2.2 (the port names are shown in *italics*).

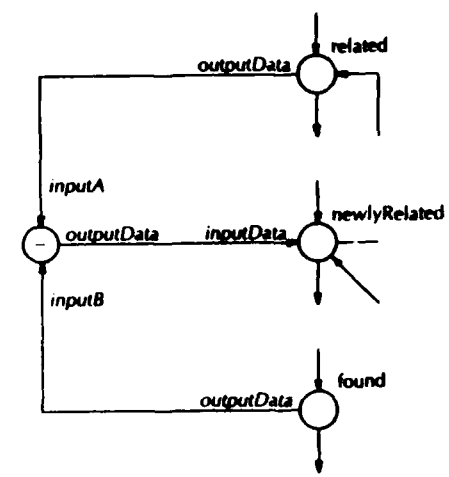


Figure 3.1 Data flow subgraph

There are two cases to be considered. In the first case, all the terminals (in this case `related`, `newlyRelated` and `found`) are selected. In the second case, the nonterminals (in the subgraph this is only -) have been selected first. The two tables below illustrate the actions of the propagation algorithm. The notation "check" means that the properties of the ports will be checked whereas the notation "*no action*" means that no action will be taken since the node on the "other end of the wire" is not instantiated. The first case is when the terminals are selected first:

from node	to node	action taken
related:outputData	–:inputA	no action
newlyRelated:inputData	–:inputB	no action
found:outputData	–:inputB	no action
–:inputA	related:outputData	check
–:inputB	newlyRelated:inputData	check
–:outputData	found:outputData	check

Table 3.2 Constraint propagation example: Terminals selected first

The second case is when nonterminals are selected first:

from node	to node	action taken
–:inputA	related:outputData	no action
–:inputB	found:outputData	no action
–:outputData	newlyRelated:inputData	no action
related:outputData	–:inputA	check
newlyRelated:inputData	–:outputData	check
found:outputData	–:inputB	check

Table 3.3 Constraint propagation example: Nonterminals selected first

3.2.2. Constraint propagation algorithm

The algorithm for constraint propagation follows:

```
; propagate-constraints takes a search node and propagates as many
; properties as possible in the data flow graph.

procedure propagate-constraints(search-node) is
   ; Loop through all the matches attached to the search-node
```



```
   forall match-node in the match-nodes
                of the instances
                of the search-node do
      ; Now search through the graph of the matched node looking
      ; for ports.
      forall parts of the graph of the match-node do
         if the part (of the graph) is a port then
            ; if the port is declared an INPUT port, then go
            ; backward through the graph (to the connecting
            ; connecting OUTPUT port).
            if the node is declared in the INPUT section then
               foreach connecting-node in
                     TraverseGraphFromNode(node, BACKWARDS) do
                  propagate-properties(node, connecting-node)
            else
            ; if the port is declared an OUTPUT port, then go
            ; backward through the graph (to the connecting
            ; connecting INPUT port).
            if the node is declared in the OUTPUT section then
               foreach connecting-node in
                     TraverseGraphFromNode(node, FORWARDS) do
                  propagate-properties(node, connecting-node)
end propagate-constraints
```

where:

```
; propagate-properties tries to propagate all the properties of the
; port-node (which points to the library via its matches)
; to the connecting-node.

procedure propagate-properties(port-node, connecting-node):
   ; First, make sure that they talk to each other at the same time.
   if the control-node of the match-node of the port-node =
      the control-node of the match-node of the connecting-node
   then
      ; loop through all the connected matches, making sure their
      ; instance is selected.
      foreach match-node in match-nodes of connecting-node do
         if the instance of the match-node of the connecting-node
            is in the instances of the search-node then
            PropagatePropertiesFromNodeIntoNode(port-node, connected-node);
end propagate-properties.
```

where:

```
procedure PropagatePropertiesFromNodeIntoNode(from-node, to-node) is
   ; This is only concerned with matching ports
   if the to-node is a port then
      if property-compatibility(from-node, connected-node) then
         AddPropertiesFromNodeToNode(from-node, connected-node)
```

```
            return success
        else
            return failure
    end.
```

and:

```
    ; property-compatibility tests to see if the properties of two data
    ; flow nodes are "compatible". This is done with a simple table lookup.

    procedure property-compatibility(from-node, to-node) is
        if compatibility-Table[from-node, to-node] = OK then return success
        else return failure
```

Note that the procedure **TraverseGraphFromNode** goes backward or forward one link in the data flow graph depending on "direction". **AddPropertyFromNodeToNode** adds the property list of the first node to the second.

3.3. Matching constraints

After a module's data flow subgraph has been matched with the program data flow graph, there are constraints that may still have to be tested. Specifically, a module may have certain use requirements that should be satisfied before the module is finally selected. A good example of this occurs in the signal processing domain where different implementations of a filter have different performance (noise, sideband suppression, Q, etc.) characteristics. These performance criteria are stated as part of the module specification and are checked before being officially matched.

As an example, take the implementation of a set using linked lists. One criterion (constraint) for selection might be "use this if the number of items in the set will exceed 100". This would be specified as part of the module specification as "(constraint (> size 100))". Examples of matching constraints can be found in the appendices. The actual checking of matching constraints is done by the search technique (see chapter 5).

3.4. Specification constraints

Specification constraints are specified by the user of the system before selection begins. These constraints reflect the goals of performance and resource usage. For example, a designer may want a design to fit in a definite amount of area or for certain procedures to be performed in a certain amount of time. The former is an example of a resource constraint (area < area-bound); the latter is an example of a performance constraint (time-for-f < time-bound). Note that both of these constraints are taken to be musts; any design created by the system *must* satisfy these constraints.

But what happens if a selection is made that violates these constraints? There are two choices: (1) throw it out and (2) try and change the design into a workable design. The "massaging" of the design is done by special purpose design operators called "critics". These will be discussed in much greater detail in Chapter 7.

Specification constraint checking, like local constraint propagation, is done with each implementation selection. Unlike local port constraints, specification constraints are binary relations between a resource and a fixed, measurable bound. The implementation of these constraints is discussed in the chapter on search (see section 5.4).

The computation of time and area bounds in the VLSI domain are complex and can only be approximated in the system because of the lack of layout knowledge. Area is currently measured by simply adding the area of the selected implementations to the current total. Real time bounds are much more complex; SIU just adds up module delays. This is not sufficient since what is really required is a notion of critical path. The lack of good timing measures is discussed in greater detail in the concluding chapter.

4. Related work in constraints

4.1. Constraints in the analysis of circuits

Sussman and his students used constraint based systems for analyzing analog circuits [SuS75][Sus7]. For example, Sussman and deKleer [deS80] used constraint equations to analyze reasonably complex circuits. They used heuristic methods (constraint propagation and symbolic algebraic manipulation) to solve for consistent solutions to these equations. Their S'N system is capable of analyzing a reasonably small circuit, such as a cascade transistor amplifier and deriving various design parameters.

These systems are analytic systems (as opposed to generative systems like SIL) and depend on the use of complex constraints to analyze the circuits.

The propagation techniques of Sussman and Stallman have been successfully used in the analysis of digital systems. Kelley and Steinberg [KeS82] have implemented a system called CRITTER that can analyze digital circuits. CRITTER uses constraint propagation to derive the timing conditions and "behaviors" of the input circuit.

Unlike the systems of Sussman and students, CRITTER uses a simple model of constraints and values. This approach is quite similar to the approach take here.

4.2. Constraints and planning

In the previous section the application of constraint propagation to circuit analysis was examined. In this section, the use of constraints in planning (the reader should consult Nilsson [Nil80] for an introduction to planning) and plan interaction will be briefly discussed. Also, the application of planning methods in specific design and analysis systems will be analyzed.

Stefik's thesis [Ste80b] (also [Ste81b] [Ste81a]) used constraint based methods in the planning and design of experiments in molecular biology. Constraints were used to detect the interference of subplans and to reduce the search space of possible new plans. These "global" constraints are similar to the "specification constraints" mentioned in the first section. Like Stefik's MOLGEN program, SIL uses constraints instead of a backtracking search.

McDermott's thesis [McD77] was directly concerned with the synthesis of elementary circuit designs from a plan description. Constraints were used to express design constraints and also to express planning preconditions. They effected the planning process by only keeping those plans that satisfied the constraints. If the constraints could not be satisfied (McDermott called this "constraint collapse"), then McDermott planned on using a planning mechanism to correct the errant plan. This is quite similar to the notion of critics as expressed in Chapter 7.

Brown's thesis [Bro76] dealt with the different problem domain of debugging circuits as opposed to synthesizing circuits. His process of "backtracing" to find bugs in a non-functional receiver is similar to constraint propagation.

4.3. Other constraint based methods

Steele's thesis [Ste80a] concerns the design and implementation of programming languages based on constraints. However, the work is of little use to the problem addressed by this thesis (partly) because Steele uses constraints as a computational paradigm rather than as a technique to cut search complexity.

4.4. Constraints and search

There is a close correspondence between satisfying local constraints, such as port constraints, and the "consistent labeling" [Mac77] problem of Artificial Intelligence. For each choice made by the selection algorithm, the choice must agree with the choices already made. Furthermore, all of the succeeding choices must agree with the choice being made. Note that every choice restricts further choices by making the problem more constrained.

Local constraints were used extremely successfully by Waltz [Wal75]. Using a labeling scheme, he used the interaction between labels in a line drawing to drastically reduce the search space of interpretations.

Mackworth [Mac77] restated Waltz' algorithm as his algorithm AC-2 and also presented other algorithms that solve the "arc consistency" problem *before* searching. After applying these "filtering" algorithms, the search space is reduced because the incompatible choices have already been removed.

Haralick and Elliot [HaE80] used an algorithm that they called "forward checking" to see if future selections will cause inconsistent labelings. They claim that forward checking will perform better (i.e., less nodes expanded during search) than most search algorithms. Nudel [Nud83] has an excellent paper detailing an analytic approach to these "consistent labeling" algorithms.

The common point of all these constraint algorithms is that the search space can be very effectively reduced by constraint satisfaction. The effect of the particular constraint approach is discussed further in the concluding chapter.

"Luck is coming your way"

Chapter 4

Matching

1. Introduction

SILI chooses implementation choices by searching through a library of template descriptions that describe the behavior (semantics) of the available modules. The object of this search is to *match* the modules with parts of the machine "generated" by the input program. The matcher described in this chapter matches the data flow subgraphs of the library modules against the program data flow graph. Since the library modules can be parameterized, a process called "binding" is used to instantiate the library templates. After binding, the library templates (now called *instances*) are ready for evaluation and search, which is covered in the next chapter.

2. Matching the library

The problem of choosing implementations begins by matching the library with the program. The data flow graph provides a reasonable representation for the matcher to work with because

(1) A data flow graph is language independent, thus isolating the definition of the library from the language

(2) The data flow graph "fits" the problem of matching parts of the machine with parts from the library.

In its purest form, subgraph matching is NP-complete. However, the problem is more constrained in SALT: each node in the data flow graph has a label. These labels allow the graph matcher to run in linear time. Assuming matches are found, instantiation of the successful matches follows. Note that failure to match every node in the data flow graph is a serious failure, since this indicates that the library fails to "cover" the data flow graph of the input program. As stated in the introduction to this chapter, the library entries are parameterized templates. This is because the modules are often of variable size (where the size depends on the values of bound parameters). For example, the size of a bit vector representation of a set is dependent on the maximum number of items in the set. Instantiation binds properties in data flow nodes to parameters in the library specification for the matched modules. After binding, it is possible to evaluate each *instance* and begin searching for the set of instances that will satisfy both the design goals and semantics of the input program.

3. Graph matching

The graph matcher works as follows: Each library subgraph has an "entry node" that the matcher uses as a start node. Furthermore, there is a table that gives the correspondence between node labels and the library entries with those entry node labels. So, the matcher process tries to match every subgraph of each module description in the library with the program data flow graph for each node in the graph. The language used to specify the match graph is nearly the same as the language used to generate the data flow graphs (see Appendix A).

3.1. Library representation

A module may have more than one data flow subgraph because a module may compute more than one result. This is the case with many modules that have control signals that dictate which function is computed. For example, a set representation may have size, add, delete and membership functions in the same module. Therefore, it follows that the library representation should reflect these different functions.

This is done by describing the module in terms of the control signal bindings. So, for each binding of the control signals of a module, there is a data flow subgraph that characterizes the behavior of the module given those control signals. For example, the representation of the parallel bit vector set representation in the example has six functions: set size, addition, deletion, membership, assignment and initialize (set to the empty set). The data flow subgraphs of this module would be:

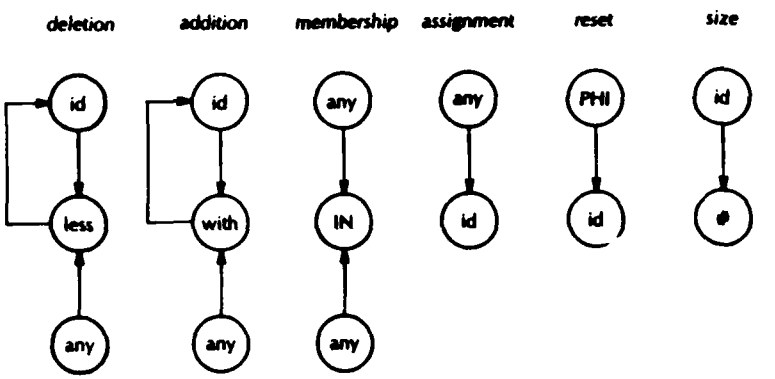


Figure 4.1 Data flow subgraph of the parallel bit vector module

The exact syntax and semantics of the data flow graph description is not of critical importance; the full details can be found in Appendix A.

3.2. Matcher operation

The graph matcher proceeds from node to node in a depth first manner by following both forward and backward arcs. It stops traversing a branch (arc) of the graph whenever

(1) A node has been already traversed (success)
(2) A node label doesn't agree (failure)
(3) A node type doesn't agree (failure)

Note that whenever a failure is detected, the whole matcher returns a failure. A success, on the other hand, indicates that the matcher has gone as far as it can go and that other paths should be pursued. It should also be obvious that no node will be traversed twice.

Note also that in case (3), node types are a direct result of property determination and declaration. The use of node types permits the matches to be restricted by type.

3.2.1. The matching algorithm

The matching algorithm (henceforth called the "matcher") assumes the existence of the following data structures:

(1) A symbol table, containing a map from symbols to nodes in the data flow graph
(2) A data flow graph
(3) A control flow graph
(4) A table with a map from symbols to possible implementations and subparts of the implementations.

The matcher starts by iterating through the symbol table. The graph matcher works backwards from the terminal nodes to the interior of the graph. Each match is recorded and associated with the implementation and the control functions that would "create" the data flow subgraph. The matcher algorithm follows, written in a pseudo set language.

```
; The Match procedure matches up the data flow nodes in the data flow
; graph with the data flow subgraphs in the library. Its side effect
; is to create match nodes that detail these matchings.
```



```
procedure Match() is
   for each symbol in the symbol-table do
      for each data-flow-node bound to the symbol do
         ; Find all the possible implementations by looking
         ; them up in the symbol to implementation table.
         foreach implement, part in the
            symbol-to-implementation-table[data-flow-node] do
               Match-from-node(data-flow-node, implement, part)
         ; Now match using the ANY nodes as well
         foreach implement in the
            symbol-to-implementation-table[ANY] do
               Match-from-node(data-flow-node, implement, part)
end Match
```

where:

```
; Match-from-node tries to establish a match starting from data-flow-node
; to the implementation "implementation" using the part name
; "part-name".

procedure Match-from-node(data-flow-node, Implementation, part-name)
   ; Each control flow node needs its own match, so ...
   foreach control-flow-node of the data-flow-node do
      ; Each "sub part" of a implementation (library module) has
      ; a graph (sub-graph). This is matched against the data flow
      ; graph. If successful, it creates a match node.
      foreach sub-graph of the implementation[part-name] do
         if Match-graph-from-node(data-flow-node,
                                  control-flow-node,
                                  sub-graph) then
            Match-Check(Create-match-node(sub-graph, implementation, part-name))
end Match-from-node
```

where:

```
; Match-graph-from-node tries to match the data flow graph
; "graph" starting from the node "data-flow-node". The matcher
; succeeds only if the nodes were used by "control-flow-node"
; The exact details of the graph matcher are omitted due to the
; dependency on the graph representations.

procedure Match-graph-from-node(data-flow-node, control-flow-node, graph)
   if the control-flow-node is in
      the control-flow-nodes of
      the data-flow-node then
         (match the graph using the description)
```

and:

```
; Match-Check tries to determine whether the match is just a new
```

```
; part of an already existing implementation.  If it is a new part,
; the it returns the instance of the old implementation, else nil

procedure Match-Check(match-node)
    ; Search all the possible matches (the union of all the matches
    ; of all the nodes in the new match-node).
    foreach other-match-node in
        union of matches of the nodes in the
        data-flow-graph of the match-node do
            ; Make sure the implementations match
            if the implementation of the other-match-node =
                the implementation of the match-node AND
                ; If so, then if the nodes of the match-node overlap (intersect)
                ; with past matches (i.e., matches already made), then return
                ; the old instance
                the nodes in the graph of the match-node intersect with
                the nodes of past-matches of the implementation of the match then
                    Add-match-to-instance(match-node, instance of other-node);
end
```

3.2.2. An example of graph matching

At this point, an example will help demonstrate the points made in the previous section.

The binary tree set implementation has the data flow subgraphs shown below:

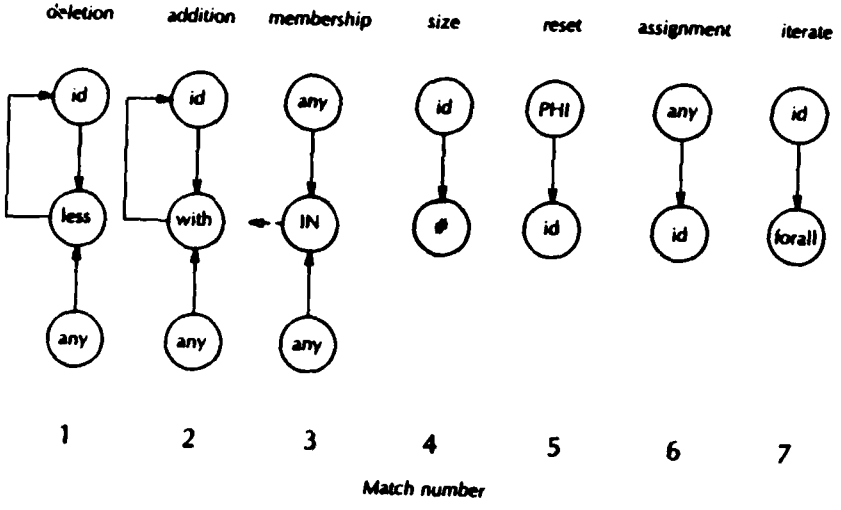


Figure 4.2 Data flow subgraphs of the binary tree module

After calling the matcher with the subgraphs of the binary tree implementation, the data flow graph of the sample program will be matched as shown in the figure below. Note that the numbers next to the nodes in the program data flow graph denote the match number of the data flow subgraphs of the binary tree implementation shown in the previous figure.

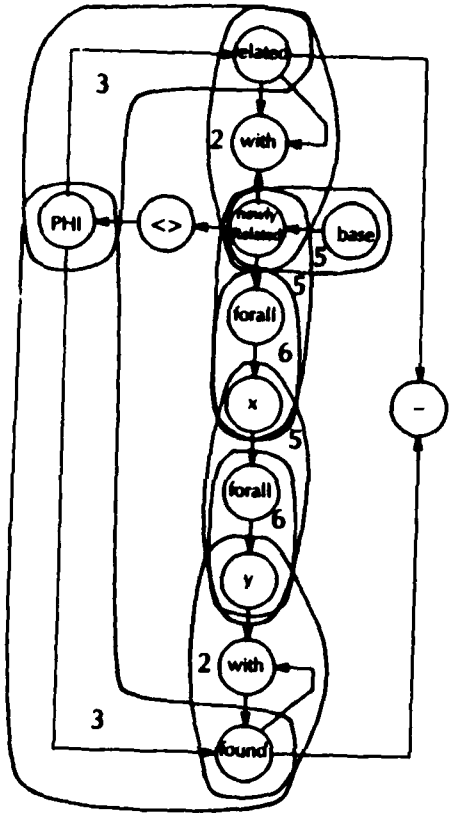


Figure 4.3 Match of the data flow graph and binary tree

4. Binding and instantiation

As stated in the first section, after matching the problem is how to instantiate the template by binding the appropriate values to the template variables. As a result of property extraction and type declaration, each identifier node in the data flow graph has type and other properties attached to it. These are the properties that are used to bind variables.

The binding between the template variables and the properties is specified as part of the library module definition in the library as a 3-tuple: the variable name, the property and the interpretation function. If the interpretation function is absent, then it is assumed to be the identity function, i.e. no interpretation is done. An example of a useful interpretation function is one that takes an integer range and interprets it as a size.

4.1. An example

Consider the implementation of the set variable x (or y) as a parallel bit vector. The critical parameter of the parallel bit vector implementation would be `wordWidth` which is the width of the implementation in bits. The relevant property of the variable is the size of the set. So, since the size is equal to the number of bits, no interpretation function is needed. Therefore, the variable `wordWidth` in the implementation template is bound to the size of the set which is an established property of the variable.

More precisely, this is done by naming nodes in the matched data flow subgraphs. After naming these nodes, then the `bind` section of the library specification can directly specify the properties in the named node. A peek at a library definition in Appendix C or D will be instructive.

5. Related work

The matching procedure described in this chapter is similar to template matching in table driven code generation. It is also similar to matching used in "idiom recognition".

These related areas will be discussed in the next two sections.

5.1. Table driven code generation

Beginning with Cattell's thesis [Cat78], there has been increasing interest in the use of tree matching in code generation. Ganapathi, et. al. [GFH82] has an overview of these techniques.

Cattell [Cat78][Cat79][Cat80] implemented a code generator generator that used heuristic search to choose the tree matching templates for the actual table driven code generator to use. His matcher was derived from his "Maximal Munching Method" (MMM) (page 37 of Cattell's thesis), which is a tree matcher that attempts to match larger trees first and then recursively tries smaller trees on the remaining subtrees.

Glanville [GlG78] used an LR parser-like system to do the tree matching. This approach is similar to Cattell's, except that it uses parser tables to do the matching.

Aho and Johnson [AhJ76] used dynamic programming to generate optimal code for expression trees (dags without common subexpressions). Their solution uses three phases. In the first pass, costs are assigned to nodes in the expression trees. These costs are derived from the code sequences that match the subtrees. The second pass divides the tree into subtrees that must have results stored in memory. The last phase actually generates the code. Their algorithm is linear in the number of nodes but it is exponential for the number of choices at each choice point.

5.2. Idiom recognition and other matchers

Geschke [Ges72] also used tree matching in his thesis work on global program optimizations. He used the notion of similarity between trees to automatically place procedures inline. The measure of similarity was made by a top-down tree walk of the two trees, comparing nodes at every branch.

Snyder [Sny82] presented an algorithm that finds and selects "idioms" (commonly found subtrees) in arithmetic expressions. The running time for his algorithm is O(n log n) for worst case recognition and O(n) for selection. While closely related to the problem of matching, idiom recognition benefits from several restrictions, which are covered in the next paragraph.

All of these algorithms share common problems.

First, all of the above algorithms choose a minimal cost template at each choice point. The problem with this strategy is that multiple paths are not explored, which may mean that other more productive paths are ignored. Second, all of these algorithms match arithmetic expression trees, not graphs. Third, all of these algorithms compute (choose) one best selection - there may be more than one, *i.e.*, there may be other different solutions with the same metric value. Finally, these algorithms generally assume that the larger the matched tree is, the better it is. This may not be true if a collection of smaller matches will do. (However, this is probably not often the case).

"Don't let doubt and suspicion bar your progress"

Chapter 5

Selection

1. Introduction

The previous chapter was concerned with how to match the library modules with the program. Once the matcher has found viable implementation choices, the next step is to somehow choose an implementation for every data flow node in the program. This chapter considers:

(1) how to search through the implementation choices
(2) how to evaluate possible implementation choices
(3) the effects of search and evaluation on the library description

2. Selection using search

2.1. Introduction

At this stage, the matcher has found matches between the library and the program. As a result of the matching, every node in the program's data flow graph should have an attached set of possible instances that involve that node. Selection is the process of choosing among instances attached to the data flow nodes. The selection procedure is also responsible for checking constraints.

2.2. The selection procedure

The selection procedure works as follows: First, the nodes in the data flow graph are sorted by the number of instances that use the node. This permits the selection procedure to start from the most "obvious" (most constrained) choices and continue to the most "complex" choices. Next, the search proceeds from node to node and for each instance attached to that node:

(1) Checks to see if this instance has already been selected by another node
(2) Checks port types of the new instance
(3) Checks for overlap of the new instance
(4) Evaluates the costs of the new instance
(5) Adds these costs to the costs of the already chosen instances
(6) Checks each new choice to see if it violates design constraints (and calls critics if it does)

The first step makes certain that the choice has not already been made. This can happen if the instance involves two or more data flow nodes and some other node has already been selected before the node being expanded. This is perfectly permissible and no further evaluation is done.

The second step checks the type compatibility of the new instance and the instances that it "talks" to. If conflicts exist, then the instance is not chosen.

The third step is necessary to ensure that the new instance does not use any of the same terminals as any of the existing instances. This prevents multiple representations for the same variable.

The fourth and fifth step evaluate the resources now consumed by the selections made so far.

The last step, step six, ensures that the new addition does not cause the generated machine to exceed design requirements. As a side effect, a possibly inefficient machine may become optimized in order to meet the design requirements given by the user of SILI.

3. Search techniques

3.1. Introduction

Search procedures can be broadly divided into backtracking and non-backtracking methods (A general overview of search techniques can be found in [Nil80]). Both of these search methods have drawbacks: Backtracking searchs (such as depth first search) can be expensive (in time costs) while breadth-first searches are exponential in space costs.

One solution, therefore, is to choose a search that can run in bounded time and bounded space. The idea behind the search used in this work is to use a modified breadth first search on an already constrained search space.

3.2. Staged Search

As stated above, breadth first search is exponential. One way to surmount this problem is to expand only the most promising nodes at any stage. Lowerre [Low76] used this in the Harpy system and called it a "beam search". (The list of available nodes is called the "beam"). Nilsson calls it a "staged search". It was originally used by Doran and Michie [Dor] in a graph traverser. The problem with a staged search is that it assumes that every step has the same cutoff factor – this is clearly not the case. When the search begins with the most constrained variable, there are very few choices. As the search proceeds, the number of choices blows up. Therefore, the idea behind the *contracting* beam search is to permit extensive branching at first and to focus (i.e., contract) the beam as the search proceeds. The purpose of the contraction is to allow as many constraints to interact as possible during the beginning stages of the search, but as the search progresses, to count on constraint interaction to bring the search within bounds.

3.3. Staged search analysis

Nilsson [Nil80] uses two measures of search performance: penetrance and branching factor. Penetrance simply is the total number (i.e., the sum) of nodes expanded divided into the path length ("levels" expanded). The branching factor assumes that pruning is *not* done at each stage of the search – of course, that is exactly what does happen during a staged search, therefore, this measure makes the most sense for a depth first search or a full breadth first search. For pruning algorithms such as the staged search, a different measure, called average branching factor can be computed. This is the total number of nodes expanded divided by the number of expanded nodes. For a search without pruning, this ratio is 1:1. Since the pruning technique used here depends on the beam cutoff and on the constraint interaction between selections, it is always less than 1.

Because the beam search operates using such fixed bounds, it is relatively easy to estimate performance parameters such as penetration and branching factor. The analysis is as follows: Assume that there are n choices to be made. Then, the depth of the search tree is n. The number of nodes expanded is dependent on both the number of available implementations and the interaction between the constraints and the implementation choices. To compute the maximum assume each implementation has k choices. Then, each step generates b (beam size) * k choices. This, multiplied by the depth gives the absolute maximum number of nodes expanded (upper bound). This is

$$N = b * k * n$$

Note that this assumes that there are *no* interactions between the port constraints of the library instances during the search. It also assumes each variable has the same number of library instances. While clearly not realistic, it is sufficient to derive an absolute upper bound.

A contracting beam search expands fewer nodes as the search proceeds. Therefore, the total number of nodes expanded, N, becomes:

$$N = \sum_{i=1}^{i=n} k * C(i)$$

where $C(i)$ dictates the size of the contracting beam. Now, let

$$C(i) = (n - i + 1) * k$$

i.e., $C(i)$ is a linear decreasing step function. Then

$$N = k^2(n^2 + n).$$

Note that in this analysis $C(i)$ is assumed to be linear with i. Such a restriction need not apply, but it does make the analysis easier.

3.4. Staged search measurements

This section discusses the measured performance of the beam search while finding a solution for the sample program. Measurements were taken of the branching factor and penetrance while the search took place. The figure on the top of the next page is the tree of choices for the sample program and the sample library. The sample library is an abbreviated version of the normal YASL library – it only has a simple set of selections (only one serial and parallel implementation for a set).

The figure on the bottom of the next page illustrates the number of nodes expanded at each level.

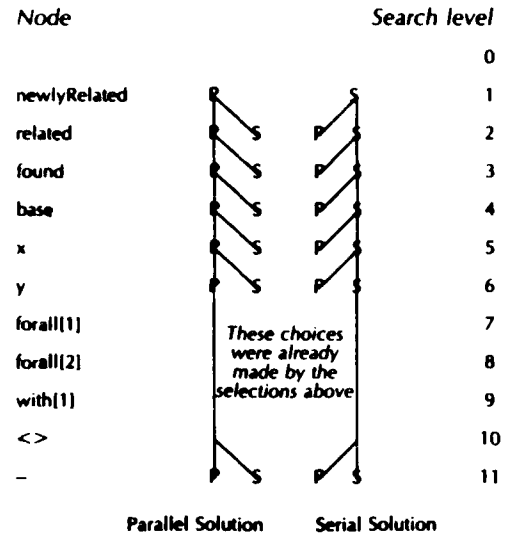


Figure 5.1 Search tree of sample program with sample library

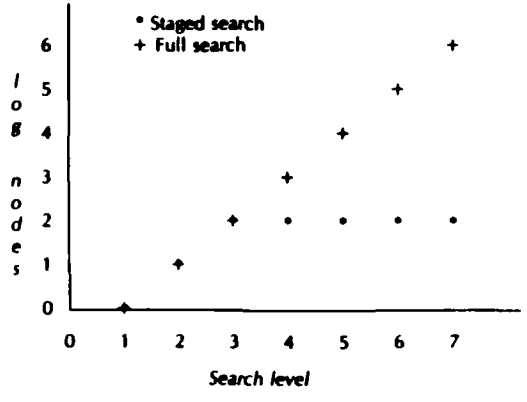


Figure 5.2 Graph of nodes expanded by level

3.4.1. The search algorithm

The following algorithm is the search algorithm used by SILI:

```
procedure Search() is
   old-node-list := nil;
   ; First, sort according to the number of possible implementation choices
   ; ("instances")
   sort search-nodes by number of instances into node-list;
   foreach node in node-list do
      new-node-list := CrossProduct(node, old-node-list);
      ; Now, sort them by the metric so that the most promising ones are
      ; at the head of the list
      sort new-node-list by score;
      switch search-type into
case CONTRACTING:
         truncate new-node-list at
            maximumBeamSize – level * beamIncrement;
         reject truncated nodes;
case STAGED:
         truncate new-node-list at maximumBeamSize;
case FULL:
      end; of switch
      old-node-list := new-node-list;
   end; of foreach
end Search;
```

where:

```
; CrossProduct does exactly what its name implies - it returns the
; cross product of the input node and the list of nodes.

procedure CrossProduct(node, list-of-nodes) is
   ; If the list is starting out, initialize it
   if list-of-nodes = nil then return node
   else
   ; Next, check to see of the instance has already been chosen
   ; (It's possible that two data flow nodes can share an implementation)
   if the instance of the node is in
      the instances of list-of-nodes then ignore
   else
   if Overlaps(instance of the node, instances of the list-of-nodes)
      then ignore
   else
      return NewEntry(node, list-of-nodes)
```

where:

```
; Overlaps checks to see of the data flow graphs of the instances
; overlap.
```

```
    procedure Overlaps(instance, instance-list) is
       foreach other-instance in instance-list do
          if nodes of instance INTERSECT with
             nodes of other-instance then
             return success
          else
             return failure
    end Overlaps
```

and where:

```
    procedure NewEntry(new-node, past-nodes) is
       ; First, check to see if the node is already there
       if node is in past-nodes then return
       else
       ; If there aren't any other nodes, then create one for sure
       if past-nodes = NIL
       then
          create new-search-node;
          score new-search-node;
          ; Here is where properties are propagated
          if PropagateProperties(new-search-node) then
             reject new-search-node;
          ; and global constraints checked (and maybe critics called)
          if ConstraintFailure(new-search-node) then
             reject new-search-node;
    end NewEntry
```

where...

```
    procedure ConstraintFailure(search-node) is
    ; To check the global (performance) constraints, check each constraint
    ; against the design. If any constraint fails, then call all the
    ; critics associated with the constraint.
       foreach global-constraint in global-constraint-table do
          if Check-Constraint(search-node, global-constraint) then
             foreach critic in critics of global-constraint do
                CallCritic(critic)
    ; Now check the global constraints again; if they're still unreasonable
    ; then return failure...
       foreach global-constraint in global-constraint-table do
          if Check-Constraint(search-node, global-constraint) then
             return failure;
       return success
```


3.5. Past work in selection

3.5.1. Automatic selection of data structures

Low [Low74] was one of the first works in automatic selection of data structures. His system chose set and record implementations for a subset of LEAP [FeR69]. His system used both analysis and simulation data to derive estimates of execution speed of the input program. The selection search algorithm was a hill climbing depth first search that pursued a minimal cost function. The search evaluation function included a function that reflected the swapping algorithm of the host machine.

Low's work has several limitations worth noting. First, he did not permit arguments of a function or operator to be of more than one type. For example, a set union operator must have compatible input data types. Also, each data type representation has only one implementation for a given operator in the library. Low did also not permit multiple representations for a given variable. This contrasts with this work, where there are many possible representations (implementations) for both operators and operands.

Low's work was extended by Rovner [Rov] to the domain of relational data structures by adding redundant representations and multiple access paths to data representations. He also instituted a two step selection scheme where implementations are constructed from primitives found in the library. Note that SEU does not do two step selections. This is because the VLSI domain encourages the design of highly compact and specialized parts. Building library parts from smaller parts could be done, but would probably be much more expensive (in both time and area) than a circuit designed specifically for that task.

Rowe and Tonge [RoT] developed another two stage refinement system that synthesized data structures from primitives. Each data type was represented by a "modeling structure". These "modeling structures" were then used to synthesize the data structure from the library. Their selection phase used a branch and bound search to do the actual selection. Their use

of a branch and bound search closely resembles the approach taken here.

Tompa and Ramirez [ToR80] developed a dynamic programming approach to data structure selection. Ramirez' thesis [Ram80] analyzes this method and other problems in automatic data structure selection.

The SETL [Sch5)] group used a technique they called "basing" [DGL79] where sets can be represented by an auxiliary data structure called a base. A base can be further specified by declarations that use the base. These declarations indicate (indirectly) the actual implementation. More recently, Schonberg et. al. [SSS81] developed an algorithm that automatically chooses bases for SETL programs. Note that the basing scheme allows runtime typing and a type analysis algorithm (such as those mentioned in Chapter 2) must be used to ascertain type information.

The basing system reflects a number of restrictions inherent in the SETL design. In particular, there are a small number of implementation possibilities and the selection techniques reflect this by the limited number of specifications for bases.

3.5.2. Automatic programming

Kant's [Kan82] LIBRA system was part of a larger automatic programming system called PSI [Gre76]. Kant used production rules to both analyze the program and generate implementation structures. The approximately 400 rules were divided into two basic categories: "searching knowledge" and "building knowledge". The "searching knowledge" was further divided into resource management and plausible implementation rules. The "building knowledge" was divided into coding and analysis rules.

Kant emphasizes the use of production rules in analyzing as well as synthesizing programs. SETL uses property extraction, declarations and occasionally user input to supply the analytic results. LIBRA also uses production rules for the selection – SETL uses a heuristic search algorithm. Also, LIBRA uses production rules to express constraints where SETL uses an explicit representation of constraints.

Kant's system was really developed for an experimental environment – her system is very flexible but also very expensive. She concluded that systems with single levels of refinement (like SETL) would perform adequately using search techniques like the ones described in the previous sections of this chapter.

4. Metrics

4.1. Introduction

At each stage of the search, an evaluation function is called to assess the resources being used at that level of the search. These functions are called "metrics" and they guide the search by "measuring" the resources consumed by each collection of instances.

The design of a metric involves two factors:

(1) fairness – the function should not permit unworkable solutions to achieve high scores
(2) accuracy – if possible, the metric function should return a value close to the "real world" resources consumed.

The last condition is required because the global (resource) constraints that the user provides are in terms that the user understands. Therefore, the system and the user must agree on the calculation of the metrics, otherwise the critics will be either called too often or not called often enough.

4.2. VLSI metrics

One problem with VLSI metrics is that they are technology dependent, i.e., an evaluation function for NMOS is not the same as an evaluation function for CMOS. Therefore, one must be careful in choosing a function that reflects the resource tradeoffs of the implementation technology. Many theoretical studies have been made of various resource bounds. While these are not directly applicable, they can provide a basis for developing a proper metric.

Most of the recent work in VLSI theory [Tho80][LiS][Bau81][ChM81] uses a complexity measure of AT^2 where A is the area of the circuit and T is the time required to compute the result. This idea was extended to the digital signal processing domain by Cappello and Steiglitz [CaS81] who used a complexity measure of ATP, where P is the period of the computation. Note that the period of a pipelined function is much less than the period for a non-pipelined function because of the higher throughput possible when the pipe is full. Therefore, this metric function favors pipelined implementations.

In Silu, wire areas are unknown until the placement and routing subsystem has been run. Therefore, it is not possible to obtain accurate figures of area consumption. As a result, the scheme behind the metrics actually used by Silu is to total the resources consumed by the non-wire portions of the machine (the modules) and estimate the wire usage.

4.3. Actual metrics

The previous chapter on binding detailed how the instantiated modules are used by the metrics to calculate the evaluation parameter. Each of these actions has an impact on the specification of the library.

As stated earlier, most library modules are parameterized so that the compiler can generate arbitrarily wide instances. The metrics also have an impact on module specification. Each module must have its height and width specified so that area can be computed. Of course, the height and width formulae can be parameterized with the module parameters and bound later. Area computation may also involve some overhead, so that must be included also.

As an example of the library specification details explained above, consider the example library. The sample parallel bit vector set representation would have the following area calculation:


```
(area   (width (times 20 n)) (height 100))
```

Likewise, timing parameters can be specified:

```
(timing (delay n))
```

After the parameters have been bound, these functions can then be evaluated and used by the metric functions.

"Very soon and in pleasant company"

Chapter 6

Machine generation

1. Introduction

The previous chapters have covered the various modules that comprise SILI. In this chapter, the underlying architecture of the generated machine is discussed. After considering the architecture, the final steps of data path creation and microcode generation will be considered.

2. Machine architecture and models of computation

Behind every machine architecture is a model of computation. Most of the machines that are in use today are von Neumann machines; they have a control store, a memory that holds *both* program and data and an arithmetic unit that is under the "direction" of the control store. If the memory is partitioned into separate areas for program and data, then the machine is known as a "Harvard Machine", after the Harvard Mark I. The next two sections present the model used by SILI and an overview of non von Neumann models.

2.1. Harvard machines

The work reported in this thesis has assumed a certain computational model. This model will be called a "maximally parallel" Harvard Machine.

"*Maximally parallel*" means there exists a unique data path between the expression computed by the right hand side of an assignment statement and the variable on the left hand side. It is maximal because for a given program, there may be any number of subtrees of the parse tree that compute the same expression but do not share operator nodes. A simple way to express this is "There is no sharing of operator hardware".

Although there are no shared data paths in these machines, the input programs presented to SILI are serial. This is due to the basic sequential nature of the input languages. There is an extensive body of literature concerned with analyzing and optimizing programs for parallel sections [PKL80]. Such techniques could be of use in constructing parallel machines. The effect of language design on architecture will be discussed further in the conclusion.

A crucial property of any sequential machine is that the machine not execute two conflicting instructions at once. "Conflicting" means that a variable is being accessed or being stored into by more than one data path. Since the control unit directs the use of the data path section of the machine, the problem becomes one of generating microinstructions that do not cause conflicts. This is easily done and will be discussed further in the section on control store generation.

2.2. Related work in non von Neumann machines

There has been considerable work in the last few years on non von Neumann architectures. In particular, reduction and systolic machines are being touted as reasonable models for VLSI implementation.

2.2.1. Data flow machines

Data flow machines are (among other things) a reaction against the serialism of register transfer machines. The serialism is due (in part) to the serial access to central memory (the

famous "von Neumann bottleneck"). In data flow machines [Den79], the results of operators are computed when the operands are present (ready). Therefore, it is possible that multiple operators can be actively computing at one time. Although some machines have been constructed, data flow machines remain largely experimental. Some of the issues involving data-flow machines can be found in Dennis [Den79] or the survey paper of Treleaven [TBH82].

How to compile "algorithmic" languages to data flow machines is another issue. Arvind [Arv79] discusses how to compile a data flow language into a multiple processor data flow machine. His paper takes extensive advantage of the fact that the language has no side effects. The effect of this "feature" on the architecture of machines will be discussed further in the concluding chapter.

Also, the work being done on the design of data flow languages such as VAL [AcD79][Ack82] reflect some basic tenets of multiprocessing such as explicit parallel operators and a lack of aliasing. Such languages would be just as useful in SIL.

2.2.2. Reduction Machines

Both control flow and data flow machines have a similar idea: data flows from sources to sinks through operators. Reduction machines are different: operations are performed by need. Reduction machines are designed to execute reduction languages [Bac78]. More work needs to be done on how to write programs using raduction languages as well as how to compile such programs into machines. Further details on some reduction machines can be found in Treleaven's survey papers [TBH82][Tre82].

2.2.3. Systolic machines

Systolic machines are essentially pipelined multiprocessors without a centralized control. Data is "pumped" from one computational unit to another with each clock tick.

Because the machines are pipelined (after a fashion), data rates are higher than equivalent sequential implementations. Kung [Kun81] and Cohen [CoZ] are vocal exponents of systolic machines. Kung points out the following features of systolic models:

- Makes multiple use of each datum
- Uses extensive concurrency
- Only a few simple cells are needed
- Data and control flow are simple and regular

VLSI designers find the regularity of systolic architectures very appealing.

Although systolic machines are a powerful use of VLSI technology (for the reasons listed above), limited work has been done on how to compile programs into systolic algorithms. Moldovan [Mol83] has recently shown how to compile loop computations using arrays into systolic arrays. Leiserson and Saxe [LeS81] present an algorithm that converts a non-systolic system into a systolic system.

It should be noted that systolic algorithms are a subclass of all algorithms; not all algorithms can be (or should be) expressed in systolic form. In particular, systolic algorithms are well suited to some computationally intensive array algorithms.

3. Machine Generation

To recapitulate, the stage is now set for the actual generation of signal paths; the program has been analyzed and the selection of implementations has been made. Machine generation begins by considering the control paths.

3.1. Control paths

The control paths of a generic machine are shown in the figure at the top of the next page. Notice how the jump signals generated by the data path section are used to control the microprogram counter. The microprogram counter is used to address the control store, which in turn generates the gating signals for the data path. A single level scheme such as this is a

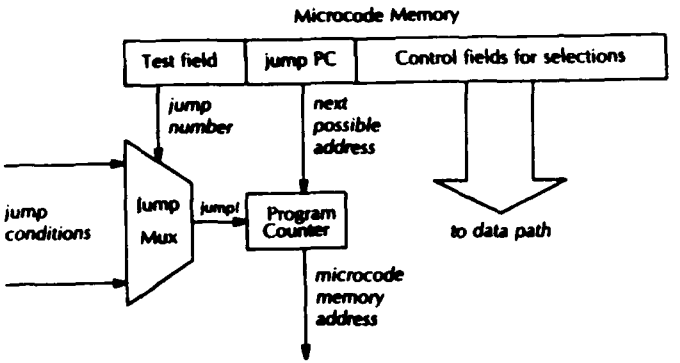


Figure 6.1 Geheric control section

very simple control store; many more complex and different controller schemes are possible. A review of microcode controllers can be found in Dasgupta [Das80]. Burke [Bur82] and Wilner and Parker [PaW81] have discussed various microstore organization for VLSI, particularly those with encoding schemes (such as that found in the MC68000).

When the data flow nodes in the program were being created by the data flow analysis procedure, they were tagged with the control flow node that was "active" at that time. For example, in an assignment statement, all the data flow nodes on the right hand side (as well as the data flow node on the left hand side) would have the name of the assignment node in the control flow graph attached to them.

Recall that the data flow nodes also have a list of instances that "involve" the instance. Therefore, it is possible to tag each instance with the control flow node via the data flow nodes. As a result of this tagging, it is now possible to tag the control flow nodes with the instances that "involve" the node.

Each control node also has a label that is used to determine how generation is done. These labels are generated by the control flow analysis procedure. Only the NODE and TEST nodes generate control fields. The remaining node labels (LOOP, LOOPBACK and EXIT) are used to control the program counter (PC) field. For more information on the labels, see Appendix A.

Note that each match node is a specific "subsection" of a library module - in particular, these matches have bound control signals. These signals are the fields that must emanate from the control store. So, control field generation is simply emitting the control bindings of every match of every instance of the implementation of a library module. The last matter in control field generation is the assignment of the microprogram counter field. Each control flow node has two pointers to other control nodes. These pointers are the "success" and "failure" pointers. Only the TEST nodes use the failure field. This field becomes the program counter field. As a default, the microcontroller assumes that the control word after the current control word will be located at the current program counter + 1.

It is important to note that the current scheme at control flow generation does not solve the problems of precedence. For example, the statement

```
a := a + 5
```

will generate a simultaneous load and store into the implementation of a. This problem is easily solved; all that is required is a procedure that detects such conflicts and moves the appropriate conflicting operation down in the control store. (In this case, it would be the store).

Notice that this control store is not compacted in any way. A useful addition at this stage would be a microcode optimizer that would move microcode fields upward in the control store. A review of microcode optimization as of 1976 can be found in Agerwala [Age76]. Fisher's trace scheduling [Fis81] is an example of more recent work.

3.1.1. Control Store Generation

The following algorithm describes the control store generation algorithm in the pseudo-set language.

```
, In ControlStoreGeneration, a.b denotes the field b of a.

procedure ControlStoreGeneration is
    foreach node in the control-flow-graph do
        word := NewControlWord();
        switch label of node into
case LOOP:
            word.success = node.success;
case EXIT:
            word.success = node.success;
case TEST:
            word.success = node.success;
            word.failure = node.failure;
case default: ; MUST be a ordinary node
            word.success = node.success;
            ; Now, for all the matches of all the instances for a
            ; given control node store the control signal bindings
            ; from the implementation.
            foreach instance in instances of search-node do
                foreach match in matches of the
                    ; This selects only the match nodes that effect that
                    , the particular instance
                    instance INTERSECT matches of the node do
                        ; Store field value (field name = control field
                        ; of match.part) by the instance. The "implementation"
                        ; is the description of the library module.
                        word.instance . control of match.part :=
                                implementation of match.implements;
```

Note that this omits the generation of the jump fields and compression of empty nodes (nodes without any control fields, just jump fields).

As an example of control field generation, consider the example program. The generated fields for one of the all parallel solutions is shown in the table on the next page.

PC	base	newlyRelated	related	found	x	y	[illegible]	next PC
0			reset					
1	load	store						
2							newlyRelated ≠ PHI	12
3				reset				
4		reset						
5		test						
6		iterate			store		iterator end	10
7					reset			
8					test			
9					iterate	store	iterator end	12
10		load	with					
11		store	load	load				
12								

Table 6.2 Microcode fields for sample program and library

4. Data paths

The previous section has shown how to construct the control paths and the control store for the machine that implements the input program. The last task is the generation of the data paths between the modules.

Data paths are informally established during selection. As each selection is made, a data path is inferred between the ports of the new selection and the ports of the selections connected to the new instance via the data arcs in the data flow graph.

It now remains to generate the final output, the net list.

5. Generating net lists

Net list generation is performed in two stages. First, the connections are made for every port in every instance (except for control ports). This establishes the data path section of the machine. Second, the output control ports of every instance are connected to the jump field multiplexer (see figure 6.1 in section 3). Now, the control fields from the control store are finally connected to the instances they control. As a last step, the control field of the jump

multiplexer is connected to the jump control field.

"Work only for the best, think only of the best and expect only the best"

Chapter 7

Design critics and Machine modification

1. Introduction

The machines that have been created so far are notable because they waste one valuable resource: area. No data paths are shared and no control fields are compressed. The intention of the design critics is to improve (or optimize) the maximal machine that was created by previous stages of the system.

The notion of critics is not unlike the optimizing pass of a more conventional compiler – except that critics do not sweep over the program like a compiler. Rather, they are called "as needed" whenever a conflict arises between user design constraints and the resources consumed by the circuit. This differs from the two previous use of critics in the planning literature.

Sussman [Sus75] coined the word "critic" to mean bodies of Lisp code that attempted to reconstitute the plan whenever a suspicious, buggy plan was added to the Conniver [McS72] data base. Specifically, critics were attached to IF-ADDED demons in the Conniver database. Sacerdoti [Sac75] used critics to detect non-working plans and to optimize plans. He applied critics at the end of each planning cycle rather than when a bad plan was detected as Sussman did. Sacerdoti also applied all of his critics at one time – Sussman only applied the critic called by the trap set in the data base by the IF-ADDED method of Conniver.

The critics proposed for use by SILI (the critics gallery was never implemented) are similar in spirit to the critics of Sussman. They are to be applied, one at a time, whenever a conflict exists between a global constraint (a resource bound) and an implementation of the input program.

2. How critics are used

As stated above, critics are used to force the resources consumed by the compiled machine to be within limits. Critics are called most often when an implementation selection is made that violates a global resource constraint.

A critic should have access to three types of information:

(1) A list of currently selected instances – This list can be used by the critic to find the implementation selection that caused the conflict. (It's quite likely that the selection that triggered the constraint violation is not the selection that really caused the constraint failure, therefore, a critic should check all the selected implementations).

(2) The current data flow graph of the data path section of the machine. One purpose of a critic is to change the underlying machine so that the resource constraint is satisfied. Therefore, the critic must be able to read (and change) the data flow graph of the machine.

(3) The exact constraint that failed. This is so the critic can determine what procedure to follow. Note that since a resource constraint can be an arbitrary expression, a critic can be called for any number of complaints.

3. Possible critics

This section suggests a list of critics that would be useful in SILI and describes how they would function.

3.1. Data path operators

3.1.1. Data path bundling

The class of Harvard Machines discussed in the previous chapter is notable for their lack of busing. Busing is used by computer architects to overcome the cost of implementing every data path between nodes in the data flow graph as a separate path. This is accomplished by sharing (time-division multiplexing) data paths under the control of the control machine. Data paths should be "bundled" together if they are infrequently used.

Tomg and Wilhelm [ToW77] presented a dynamic programming solution to bus allocation. While their algorithm is optimal, it involves an expensive combinatorial search. A simpler busing algorithm was developed by Tseng and Sieworek [TsS81]. Their technique creates buses one at a time by trying to assign as many data paths to a new bus but without introducing delays (and reducing concurrency). Tseng and Sieworek's procedure would be extremely useful to SILI.

Note that while busing reduces wiring area, there may be additional costs of adding bus drivers if the units that are bused do not have bus drivers. Note also that the busing critic does not know about "passthrough" functions. Passthrough functions are operations that turn a functional unit into a straight through connection, i.e., no operations are performed and data is "passed through" Such functions make "indirect paths" (Tomg and Wilhelm's term) possible.

3.1.2. Functional unit sharing

Besides not sharing buses, the uncriticized machine doesn't share functional units either. Functional unit sharing can take place when two (or more) functional units are identical instances; i.e., their parameters are the same.

Note that sharing a functional unit may have a time penalty – an operation that was formerly performed in parallel may now have to be performed serially because the functional unit must be shared among two computations. This type of tradeoff is very difficult for the system to make.

The sharing of functional units could be performed by searching through the selections and trying to find two selections that are equivalent, i.e., the properties attached to the ports are *identical* and that the units are of equal size.

When the decision to share a functional unit has been made, the shared functional unit must have multiplexers introduced on the inputs. The space tradeoff for this is the cost of the multiplexers versus the cost of the additional unit. In all but the most extreme cases, the cost of the multiplexers is very small compared to the cost of the additional functional unit. The exact algorithm is as follows:

```
; FindAndShare takes the whole list of "instances" and tried to
; find units that can be shared.

procedure FindAndShare(instances) is
   foreach first-instance in instances do
      foreach second-instance in instances – first-instance do
         ; If the two instances are instance of the same library module
         if the implementation of the first-instance =
            the implementation of the second-instance AND
         ; and the parameters match...
         BoundParametersMatch(first-instance, second-instance) AND
         ; and the instances aren't used at the same time...
         control nodes of first-instance INTERSECT
         control nodes of second-instance = NIL then
            Create a multiplexer for all the inputs of first-instance
            Connect the inputs of the first-instance to the multiplexer
            Connect the inputs of the second-instance to the multiplexer
            Connect the outputs of the first-instance to the destination
                  of the second-instance ;(this assumes tri-state busing)
            Get rid of second-instance
end

where:

procedure BoundParametersMatch(first-instance, second-instance) is
   foreach parameter in the library description of first-instance do
      if the value of the parameter in first-instance ≠
         the value of the parameter in second-instance then
            return false;
   return true;
end
```


3.2. Pipelining

Register transfer machines have problems with data rate. This becomes apparent when one considers that data must flow from the input node to the output node over a number of computational steps (sequences). So, at a minimum, the output data rate is proportional to the length of the microprogram. (This assumes no loops). At worst case, the output data rate is proportional to

$$non - loop + \sum_{i=1}^{L} loop[i]*n[i]$$

where $loop[i]$ = loop section[i]
and $n[i]$ = maximum number of iterations for loop section i
and L = number of loops

The output rate of a register machine can be improved by introducing latches at the beginning of each stage. Hence, partial results of a computation can be held in several stages, similar to a production line. An introduction to pipelining can be found in Ramamoorthy's survey [RaL77]. Kogge [Kog81] is an extensive reference.

Pipelining can be easily introduced into the machine by the introduction of latches (called "staging latches") at the input and output ports of every instance. The control of the staging latches can be done easily by the control store. However, there are several problems with pipelined machines. First, conditional statements cause branches, which break up the data flow. Second, feedback loops in the machine can cause the machine to wait for data to be fed back. Third, loops in the microcode can introduce delays (and subsequent loss of throughput) by keeping functional units busy that are fed by data paths above the loop. Such delays must be compensated for by memories such as queues.

Leiserson and Saxe [LeS81] have developed an algorithm that converts semi-systolic machines into fully systolic machines. Their procedure makes use of the notion of adding delays ("retiming") to the arcs that connect operators. A similar form of retiming could be useful in the transformation of register transfer machines to pipelined machines.

Although no pipelining critics were implemented, the system could have benefited from the use of both an algorithm for the insertion of staging latches and shimming delays. Such critics are a necessity in the digital signal processing domain where speed is often of the utmost importance.

3.3. Pinout limitations

Although VLSI circuits are increasing in complexity, there are fundamental physical limitations that prevent the implementation of certain circuits. Pinout limits are an example of such a physical limitation. Pinout limits are a result of packaging limitations. Any circuit that is designed by a VLSI compiler must not exceed the maximum number of pins for a given packaging technology. The fundamental technique for avoiding pinout problems is to multiplex pins. This is commonly done in many commercial microprocessors. Of course, this has its price – it limits the data rate through the multiplexed pins. The pinout critic would be called when the number of pins exceeds the package count. The number of pins currently used is simply the number of signals (ports) without attached ports.

3.4. Control section operators

3.4.1. Optimization

As pointed out earlier, although the data path can operate in parallel, it is strictly limited by the serial nature of the control machine. Recall that each microprogram word "represents" a statement in the input program. It is possible that some fields in the control store may "lie fallow" which the remaining fields are used in the computation of the statement. These unused fields may be used in subsequent computation, so it pays to try and pack these fields as tightly as possible. As mentioned in the previous chapter, This brings in the whole realm of microprogram optimization. Dasgupta [Das80] and Agerwala [Age76] have fine reviews of some of the techniques in use by microprogram optimizers. Davidson, et. al. [DLS81] performed some experiments on compacting horizontal microcode (such as that generated by SILI). The application of their techniques to the output of SILI would be extremely advantageous, as control store compaction cuts area of the control store ROM.

3.4.2. Field encoding

Another possible optimization is to encode several control fields together. This is particularly useful when there are many one bit control signals of which only one is active at a time. If this is the case, then 2^n signals can be encoded as n wires plus the overhead of decoders. These decoders are placed at every use of the encoded control signals. Saunders [Sau79] describes a similar optimization that can be performed when constructing specialized interpreters.

4. What to do when critics fail

Critics can fail to obtain their objective. The simplest case of this is when a critic is unable to make any improvements in the machine. This may occur when a machine has already been optimized and another critic is called. Unfortunately, the way out of this dilemma lies directly with human intervention. In particular, the user can be informed of the inability of the system to make any improvement and the "suggestion" is made to change the resource constraint. After changing the constraint to a more reasonable value, the system is free to proceed.

"Everything will now come your way"

Chapter 8

Implementation, Results and Conclusion

1. Introduction

First, this chapter will discuss the actual implementation and results of the ideas presented in the last 6 chapters. Next, areas for future research will be explored and finally, the conclusions will be presented.

2. Implementation

SRLI is organized along the lines shown in the figure on the next page. Solid lines denote data flow; dotted boxes denote unimplemented sections. The labels on the arcs are the names of data formats.

Before SRLI can process the input program, the various language dependent files must be read in. SRLI is designed to be language independent – the syntax and semantics of the language are defined by files that represent the parsing rules (productions), the data and control flow "equations", the property propagation table and the implementation library.

Briefly, SRLI runs as follows: the input program is scanned and parsed by a recursive descent parser. The output of the parser is an ordinary parse tree. The parse tree is used as an intermediate form for several stages of analysis. The first action after parsing is control and data flow analysis. The analysis algorithm is described in Appendix A. After this is completed, both the data flow and control flow graphs have been constructed. Next, the parse tree is traversed and declarations of types and other properties are attached to the terminal

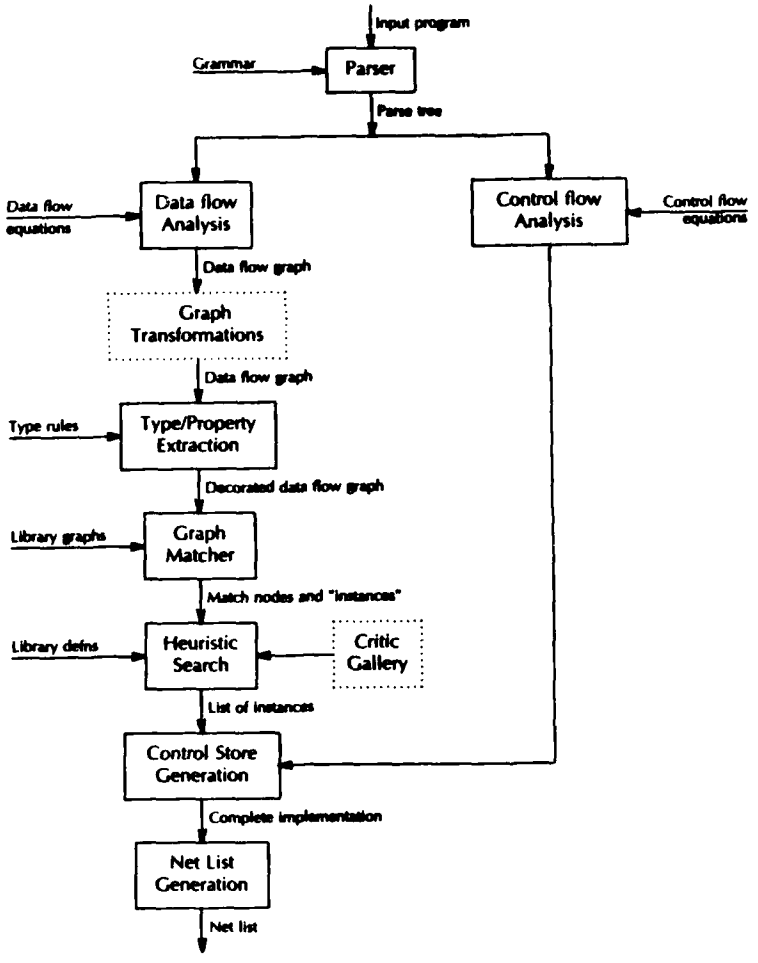


Figure 8.1 Detailed block diagram of system organization

data flow nodes. Note that the rather baroque type declarations of both YASL and CLASP are meant as a substitute for more involved property extraction. Next, property extraction is done and properties are propagated to the non-terminal (interior) nodes of the data flow graph.

Matching uses a table of correspondences from data flow nodes to possible implementations that use that node. Matching tries to match the data flow subgraphs of the implementations in the library starting from each node in the data flow graph. The selection stage weighs the costs of making each selection and also checks the port and specification constraints.

Note that critics may be called at any stage of the search, hence there is a dashed line to the "critics gallery", which is intended to be a collection of LISP code. Finally, the remaining implementations are given to the control store generator, which creates the control store and assigns the control fields. The final output is the net list generated from the implementations.

The implementation was written in Franz-Lisp, a MacLisp dialect (in turn a descendant of Lisp 1.5) that runs on the VAX-11 series computers. The program occupies 475 pages of memory before compilation begins. The YASL program used as an example ran interpretively on a stand alone VAX 11/750 in 60 minutes of online time and 55 minutes of compute time.

3. Results

The example program generated two solutions (the fully serial and fully parallel solution) using a full library and a staged beam search.

A full annotated run of the sample program and a larger YASL program is shown in Appendix C.

The ultimate goal of this work, as elucidated in the introduction, was to enable an unsophisticated user to generate a VLSI circuit that executed the user's program and also met the user's established design requirements.

SILI meets these goals through its exploitation of various constraint based methods and heuristic search. However, a thesis often introduces more problems than it solves; this work is no different.

4. Directions for future research

4.1. Semantics

The semantics of most programming languages are defined informally through the use of procedures called "semantic routines". Only recently have more formal methods such as denotational semantics been used to describe the semantics of languages. These "semantic routines" are called during syntax directed translation. SILI is different: the semantics of the library modules are partially described by a data flow graph. The matching procedure essentially states that a piece of the program and a subgraph are equivalent – both in terms of the graph and the semantics "expressed" by the data flow subgraph. What this means is that the nodes generated by the data flow analysis procedure have a particular syntax and that it is the matching that defines the meaning (or semantics). Of course, there's much more to the semantics of hardware or VLSI (or programs for that matter). A much more extensive effort is needed to define the semantics of hardware (broadly construed).

4.2. Critics

As in Sacerdoti, the use of critics in SILI is a replacement for a more precise semantics of optimization.

> "The constructive critics ... were developed in an ad hoc fashion. No attempt has been made to justify the transformations that they perform or to enable them to generate all valid transformations." (Sacerdoti [Sac75], pp. 126)

This is due to the lack of semantics of hardware optimization. This is another area ripe for exploration.

4.3. Lack of procedure calling mechanisms

The astute reader may have noticed that there hasn't been a mention of procedures. This is not serious if the depth of procedure calling is not great – the immediate solution is an on-chip stack that can be part of the control section. However, for recursive procedures, this becomes a much more serious fault. One solution is to move the stack off-chip – but this introduces the delays associated with off chip memory. Another solution is to try and compile the recursive function into a network of machines. This notion will be explored in the next section.

4.4. Interaction of machines and languages

Programmers know that languages heavily influence their programming style. Likewise, languages exert heavy influence on the machines that can be generated from programs written in them. In particular, languages with assignment introduce the notions of global and shared state. This restricts the implementation by reducing the amount of parallelism in the resultant machine. This is widely recognized and efforts are being made to change these notions. For example, Arvind [Arv79] has shown how to compile a language without side effects into an array of machines. VAL [AcD79] is a language designed for execution on data flow machines without global state. There should be more work done in how to compile such languages into machines.

In fact, there should be more work done is how to compile languages into machines of any form. One of the few works on this topic is Wand [Wan82]. He discusses the automatic creation of machines from a denotational description of the language. The system uses combinators which become the "instructions" of the machine.

4.5. Memory hierarchy

4.5.1. Registers

Although not explicitly stated, this work has assumed a simple model of memory hierarchies. For example, there are no local registers, as commonly found on most machines. Local registers are used to hold intermediate results of computations such as common subexpressions. They are used to save time by not storing results in the more costly external memory. It would be possible to introduce registers as a side effect of functional unit sharing. Such actions are not done in SILI.

The graph coloring has been useful in register allocation for determining when register "spilling" should be done [CAC81]. A derivation of such an algorithm might be useful for planning the location of registers on a chip.

4.6. External memory

Programs rarely use a small amount o memory. Any VLSI system that is designed by a silicon compiler must plan on using an off chip memory for some (possibly all) applications. As it stands now, SILI is not cognizant of any notion of off chip memory. This is because the time-space tradeoffs of going on and off chip can be done using the existing methodology of search and evaluation. In particular, the use of an external memory offers the space cost of just the drivers and logic, not the memory array. Likewise, the time penalty is the cost of going off chip plus the memory access time. Both of these parameters can be easily expressed given the existing descriptive mechanism. As an example of this, Appendix C has a linked list set implementation that uses external memory.

However, there's more to the problem. What happens when two or more operator implementations use an external memory array? This is similar to the problems faced by multiprocessor access to memory. There are basically two solutions:

(1) Divide the memory into two, either by separating the memories or by using mapping.

(2) "Synchronize" the algorithms used by the implementations so that they cooperate (for example, by sharing memory allocators),

4.7. Timing measurements

Unfortunately, SILI lacks a good timing measurement subsystem. This was strictly due to the amount of effort spent in describing and analyzing the timing of the generated circuits. As it stands now, SILI adds up the "delay" times that are specified as part of each implementation specification. This should be replaced with a timing analysis subsystem that uses such measurements as the delay from statement to statement or the delay of a loop. Systems like those described by Cohen and Zuckerman [CoZ] or Ramshaw [Ram79] could be extended to cover such timing calculations. A complex timing analysis subsystem should be part of any future silicon compiler.

4.8. Matching computation rates

Little mention has been made of the problem of differing computation rates, particularly with pipelined implementations. When two implementations are connected and they have different periods (*not* delays), then some attempt must be made to match the difference. Typically, queues and caches are introduced to solve these differences. A truly complete system would automatically introduce such interfaces.

4.9. Types and type generators

An underlying current of this work has been the use of types in Very High Level language design. Specifically, recall that the data flow graph matcher uses mode typing as part of the matching procedure. This enforces the notion that the implementation of data is divided by type.

However, there are many shortcomings and the solution is not immediately obvious. In particular, consider the problem of type generators (ALGOL 68 calls them "type constructors") for the generic data type "set". An example of the usage of such a constructor would be a "set of integers" or a "set of floats". SILI attempts to match an implementation of a data type with a module that implements that data type. This is known as a "one step refinement." Unfortunately, this makes the designers task harder – the designer must create a new module for every new type! A better scheme would be the use of type generators (constructors) – unfortunately this is very hard. The difficulty lies in the creation of a circuit that can be extended across differing base types (for example, useful for both integers and floats).

4.10. Making the design debugable and testable

Programs seldom work the first time; unfortunately, digital circuits aren't much different. Therefore, some provision should be made for the insertion of hardware that makes the testing and debugging of a design easier. While possibly not a standard option, these additions should be available if the user requests them.

Sproull and Frank [FrS81] have an overview of some techniques that could be used by a silicon compiler as well as a circuit designer. Also, Williams and Parker [WiP83] have a review of design techniques that increase the testability of VLSI design.

5. Conclusion

This work is one step toward the ultimate goal of a system that compiles a program to a description of an integrated circuit. This goal has been achieved by using techniques from Artificial Intelligence and conventional compiler theory and practice. The work reported in this thesis has shown that:

- Compiler techniques can be used to generate machines for VLSI implementation from programs

- Very High Level Languages can be used to hide the implementation complexity of VLSI design
- Constraint methods are useful and applicable to the VLSI problem domain
- Heuristic search and constraints can be successfully used to choose implementations with differing costs
- Resource constraints can be used to control the optimization of the design by calling specialized code

"Your mind is filled with new ideas. Make use of them"

Appendix A

Flow Analysis Technique

1. Introduction to flow analysis

The use of flow analysis in compilers is quite common. Flow analysis can be divided into two parts: control flow and data flow analysis. Control flow analysis is concerned with how the program (or more precisely the program counter) changes from statement to statement. Data flow analysis is concerned with how data flows from variable to variable.

In the past decade, there has been a considerable body of literature published that exposes the more theoretical nature of data flow analysis (including its intimate connection with lattice theory [Kil73]). An introduction to the use of flow analysis in compilers can be found in Aho and Ullman [AhU77]. Kennedy [Ken81] has a fine overview of the existing techniques.

To review briefly, data flow analysis can be divided into two categories: high level and low level. High level analysis begins with a parse tree or some other "high level" representation. Low level analysis uses "lower" level representations such as connection matrices.

The output of either form of analysis is a low level structure such as a matrix of USE/DEF bits.

There are, however, limitations on the present collection of flow analysis techniques. First, they are strongly language dependent. There has been limited work done on making these techniques table driven. Donzeau-Gouge's [Don81] work on generating data flow from denotational semantics is a first step. Second, the output of flow analysis is generally used for optimization, not for the generation of the data path section of a machine.

The flow analysis technique described here was created to solve these two problems. The analysis procedure accepts a description of the control and data flow "equations" for each left hand side of a production in the language grammar. Each "equation" uses the parse tree of the input program as a source of data and control. The final output of the procedure are the control and data flow graphs for the input program.

2. A description of the technique

2.1. Introduction

The analysis technique is a constructive one, that is, the graphs are constructed incrementally as the analysis proceeds. The graph is synthesized by a graph interpreter that interprets a special language designed for flow analysis. This language will be described further in section 2.4.

2.2. Control flow and data flow: differences and similarities

At first glance, there appears to be little difference between a control flow graph and a data flow graph. They both are directed graphs, possibly with cycles and they both have nodes with multiple edges leading in and out. However, there are a number of subtle differences that will arise when the actual interpreter is implemented. These differences will be apparent as the primitives as the various are discussed in section 2.4.

2.3. The basic idea

The basic idea behind the flow analysis technique is to use a graph grammar to construct the flow graphs. This is similar in spirit to Kennedy, Farrow and Zucconi [FKZ76] who used a graph grammar to analyze a restricted set of flow graphs. The primitives of the graph language are the terminals of the language. The stack of the interpreter acts in much the same fashion as the stack of a parser. The interpreter is directed to interpret new branches of the parse tree by primitives in the language.

2.4. Primitives

There are 15 primitives; some of them are restricted to control flow and some are restricted to data flow. The independent primitives are:

- Attach-head <expression1> <expression2>
  Forges a connection between two nodes. This is the fundamental primitive for forming links. This returns the first expression (node1).
- Attach-tail <expression1> <expression2>
  This is similar to attach-head, but returns the result of evaluating the second expression (expression2).
- Follow <expression>
  is the mechanism that introduces the flow of control; follow needs a field of the parse tree to pursue, i.e., (follow car) says to recursively call the interpreter with the car of the parse tree.
- Loop <expression>
  begins a loop. Each loop has a body which is the following expression.
- Loopback
  is a way to create an arc back to the nearest loop. Nearest means that loops are kept in stack order. Note that this eliminates naming, but at a cost: arbitrary exits and loops from loops are not permitted.

The next section discusses the primitives specific to data flow analysis.

- Do <expression> ... <expression>
  is similar to the ALGOL-60 BEGIN ... END pairs; technically it is not needed - it is mainly a syntactic device.
- Node? <label> <node name> <symbol table>
  is used to search symbol tables; if the node is not in the table, then the node is created and inserted in the table. The first field is the label to be given to the node. The second field name name to be searched for or created. The third and

last field is the name of the symbol table. This permits multiple symbol tables.

- Node! <label> <node name> <symbol table>
  is identical to Node? except that nodes are created without being looked up. This creates multiple nodes for a given name. This would be used for the creation of interior nodes in the data flow graph.

The following are the primitives that are specific to control flow analysis. They are:

- Enter <expression> *
  returns the name of the entering node of the expression, i.e., the node without a predecessor. This is possible because each control flow node has a link both forward and backward. Enter chases the backward links until the field is NIL.
- Exit <expression>
- Exits <expression>
  are two versions of the same primitive. Exits returns the list of multiple exits given a single node. An exit is defined as a node without successors. Exit is similar but returns only one node. If more than one exit is possible a bug trap is called.
- Fork <name> <success> <failure>
  creates a node with the name <name> and two exits; a "success" exit and a "failure" exit.
- Join <name> <expression> ... <expression>
  joins together a collection of nodes into a new JOIN node (with the name <name>).
- Node: <name>
- Follow: <expression>
  These are the links between control flow and data flow analysis. Node: creates a node with the name <name> and then transfers control to the data flow analysis routines. When the data flow analysis is completed, control returns to the expression that called the Node:. Follow: is identical to Node: except that a new control node is not created. Note that these commands are needed to "synchronize" the control flow and data flow analysis routines. In particular, the data flow nodes must have the control flow nodes that were "active" when the nodes were generated. This is used by the microcode generator described in Chapter 5.

2.5. Power of the method

The data flow analysis and control flow analysis methods described here are powerful enough to handle the demands of a restricted set of "realistic" languages. The control flow analysis routine is limited by the `loop` and `loopback` nodes. Although not implemented, a `loopforward` primitive would be possible and would extend the generative power of the technique to cover loops with arbitrary exits.

Although somewhat limited, this method is powerful enough to cover the "structured flow graphs" (a subset of the "semi-structured flow graphs" of Farrow, et al. [FKZ76]) of

Böhm and Jacobini [Boj66]. Of course, the addition of the `loopback` primitive extends the range of graphs generated. A `loopforward` primitive would extend the class further.

3. Example

Consider the example of a `while` statement. The control flow specification for this statement (in YASL) is as follows:

```
(whileStatement
  (
      (loop
        (attach
          (follow cadr)
        (fork TEST
              (enter (loopback (exit (follow cddadr))))
              (node EXIT)
        )
         )
      )
  )
)
```

Here, the cadr branch of the parse tree is the boolean expression, while the cddadr branch of the same tree is the top node of the statement. Note how the `enter` and `exit` primitives are used to get both the top and bottom nodes, respectively. Also, note how the `loop` and loopback `primitives are used to get the looping of the while statement.`

4. Conclusion

The method described here met all the goals set before as described in the first section. It is language independent, simple and reasonable efficient. However, there are some interesting new directions:

(1) Is it possible to automatically generate the "equations" given the definition of the semantics of the language (such as denotational semantics)? The answer is probably yes, but the work remains to be done.

(2) Exactly how efficient is this method? In terms of space, this method clearly uses a fair amount of space (mostly on the stack). In terms of time, the method is relatively simple. An exact measurement or calculation would be interesting.

"Leave your boat and travel on firm ground"

Appendix B

Library format

1. Introduction

The libraries for both YASL and CLASP are specified by giving a definition of each module in the library. The module definitions have two parts. The first part contains the specifications that are generic to the module (such as port declarations and resources consumed). The second part contains the various parameters of the control section dependent "parts". Both of these sections are discussed next.

2. Generic definitions

The generic section of a module definition contains eight subsections. It begins with the declaration of the parameters of the module. These are the parameters that are *bound* during the binding process. For example, the width of a bit vector set representation can be declared as follows:

```
(variable bitWidth)
```

The next two declarations declare ports to be either input and/or output ports. (Note that it's quite possible that a port is bidirectional and hence can be labeled as both input and output). The name of the port must be followed by the width of the port. So,

```
(inputs (inputPortName bitWidth))
```

(outputs (outputPortName bitWidth))

Next, if a port is to be connected to the control store, then it must be declared as a control port. The declaration looks as follows:

```
(control (controlPortName bitWidth))
```

After declaring the ports, the properties of the ports must be declared. These are the properties that are needed for port constraint propagation. The following declaration declares "inputPortName" to have "parallel integer 2s-complement" properties:

```
(properties (inputPortName parallel integer 2s-complement))
```

Lastly, the generic declarations must state the resources consumed by the implementation. Currently, the resources are limited to area, time and power. What follows is a sample declaration for a module with a width of bitWidth (declared by the parameter section described above), a height of 20 lambda and an overhead of bitWidth * 5 lambda. (All area and length metrics are given in lambda which is the minimum feature size [MeC78]). The time is given in nanoseconds. Note that there are two time parameters: delay and period. The power figure is given in milliwatts.

```
(area   (width bitWidth)
        (height 20)
        (overhead (times bitWidth 5)))
(time   (delay (lookup delay))
        (period))
(power (times bitWidth 100))
```

3. Function specific declarations

Each module may have several functions. These functions are specifed by declaring five parts. The first part declares the "name" of the function. The second part declares the control signal bindings that cause the function to be performed. For example, if the control port "operation" is bound to a two, then the declaration would be:


```
(control (operation 2))
```

The third part is critical to the operation of the system. This is the declaration of the data flow graph to be matched with the data flow graph of the input program. The form of the graph to be matched is identical to the description of the data flow graphs (see the appendix on data flow graph generation for more details) except for a few details. In particular, it is necessary to identify the ports of the graph. The other change requires that when nodes are matched, types are matched also (if a type field exists). For example, consider a port that is also a node. This node can have a type to match as well. The following example demonstrates both these features.

```
(port inputPort (node IDENTIFIER SET INTEGER))
```

This is a single node graph with the type "set of integer" attached to the node, which is in turn named "inputPort".

The next declaration declares the timing parameters for the function. This is because each function may have differing timing parameters. Note that this can create a problem when the timing parameters of a module's functions differ. The final part of a definition is the binding fields. Binding is described fully in Chapter 4. Briefly, the form of a binding declaration is as follows: The first field is the name of the variable to be bound. The second field is the name of the port to lookup the property whose field will be bound to the variable. The third field is the name of the property and lastly, the fourth field is the optional interpretation function. For example, the declaration:

```
(setSize inputPort size)
```

will bind the variable "setSize" to the size field of the node that is matched to the port called "inputPort". An example of a binding function with an interpretation function is given below:

```
(bitWidth inputPort range range-size-in-bits)
```

Here, range-size-in-bits is a function that returns the log (base 2) of the ceiling of the range of bits.

4. Library syntax

The Backus-Naur Form (BNF) for the library follows below:

```
library ::=
    name components
components ::=
    name ( NAME fields ) components
    nil
name ::=
    IDENTIFER
fields ::=
    ( VARIABLE parameterList )
    ( OUTPUTS portList )
    ( INPUTS portList )
    ( PROPERTIES propertyList )
    ( AREA areaList )
    ( TIME timeList )
    ( POWER expression )
    ( PARTS partsList )
parameterList ::=
    IDENTIFIER id-list
    nil
id-list ::=
    IDENTIFIER id-list*
id-list* ::=
    , id-list
    nil
portList ::=
    ( port-name width ) portList
    nil
port-name ::=
    IDENTIFIER
width ::=
    NUMBER
    IDENTIFIER
propertyList ::=
    ( signal-name properties )
signal-name ::=
    IDENTIFIER
properties ::=
    property-name properties
    nil
property-name ::=
    IDENTIFIER
areaList ::=
```



```
    ( WIDTH expression ) areaList
    ( HEIGHT expression ) areaList
    ( OVERHEAD expression ) areaList
    nil
timeList ::=
    ( DELAY expression ) timeList
    ( TIME expression ) timeList
    ( PERIOD expression ) timeList
    nil
partsList ::=
    ( part-name partList ) partsList
    nil
part-name ::=
    IDENTIFIER
partList ::=
    ( CONTROL signalList ) partList
    ( GRAPH graphDescription ) partList
    ( TIMING timeList ) partList
    ( BIND bindList ) partList
    nil
bindList ::=
    ( parameter-id port-id property-spec
                property-interpretation-function ) bindList
    nil
parameter-id ::=
    IDENTIFIER
port-id ::=
    IDENTIFIER
property-interpretation-function ::=
    IDENTIFIER
property-spec ::=
    property-name
    ( FIRST property-name )
    ( SECOND property-name )
    ( THIRD property-name )
property-name ::=
    IDENTIFIER
expression ::=
    s-expression
```

"The night life is for you"

Appendix C

Yet Another Set Language

1. Introduction

YASL is another set language. It is descended from VERS2 [Ear74] and SETL [Sch5)]. It differs from SETL significantly in that it does not have all the trappings of a full set theory (in particular, the notions of mappings and functions). It also lacks the relational power of VERS2. However, the language has a full complement of set operators, including existential and universal quantifiers. Unlike SETL, YASL is a statically typed language (i.e., types are decided at compile time).

YASL programs resemble programs written in "conventional" algorithmic languages such as ALGOL-60: The data type declarations of the variables come first and the body of the program follows.

The next section informally discusses the syntax and semantics of YASL. Following the description of YASL, the library is included. This is the actual library used by the demonstration program. Lastly, this appendix concludes with the transcript of the example program (shown in figure 1.1). being compiled by the system.

2. Description

2.1. Introduction

The syntax of YASL is described in the next two sections using Backus-Naur Form (BNF). The reader unfamiliar with this should consult [AhU77] for an explanation. All terminals are in upper case; non-terminals are in lower case. Each separate line is a production. *The symbol "nil" indicates an empty (epsilon) production.* These productions match the next token under all conditions.

2.2. Lexical Input

The input to the parser comes from the scanner. The scanner has some simple rules for scanning tokens. These are detailed below:

(1) Identifiers are scanned by detecting an initial alphabetic (a-z, A-Z). This is followed by an arbitrary string of alphanumerics (a-z, A-Z, 0-9).

(2) The number scanner assumes that all numbers are integers. While this doesn't implement the full YASL language, it was sufficient for the demonstration program.

(3) The operators and delimiters of YASL are composed of special characters.

(4) YASL uses the MESA [MMS79] comment style: anything beyond a double hyphen (--) is ignored until the end of the line.

2.3. Declarations and scope rules

All the variables in the program must be declared. If a variable is not declared, then the selection phase will be unable to find any implementations that "cover" the variable since the selection phase depends uses type information to separate the implementations. (For further information on the use of type information, see Chapter 4).

Unlike ALGOL-60 and its derivatives, YASL does not have scoping rules. This strictly due to the amount of effort expended in constructing the symbol tables. There is nothing inherent in the lack of scoping (except laziness).

2.4. Declarations

YASL declarations serve two purposes: first, they specify the type of the variables. The second purpose is to give hints to the system about various parameters, such as the size of sets and the range of integers.

```
declaration ::=
   setDeclaration
   tupleDeclaration
   integerDeclaration
   floatDeclaration
   characterDeclaration
integerDeclaration ::=
   INTEGER optionalRange : idList
floatDeclaration ::=
   FLOAT optionalRange : idList
characterDeclaration ::=
   CHARACTER : idList
optionalSize ::=
   WITH SIZE NUMBER
   nil
optionalRange ::=
   WITH RANGE BETWEEN NUMBER AND NUMBER
   nil
```

Besides basic type declaration, declarations can also be used to give hints to the selection system about the size of the various elements.

Here are some examples of declarations using the basic types:

```
integer with range between 0 and 100 : x;
float : z; -- No hints in this one!
```


2.4.1. Set and tuple types

The declaration of sets and tuples use the basic types as subtypes. The syntax of set and tuple declarations are as follows:

```
setDeclaration ::=
   SET optionalSize OF declaration
tupleDeclaration ::=
   TUPLE optionalSize OF declaration
optionalSize ::=
   WITH SIZE NUMBER
   nil
optionalRange ::=
   WITH RANGE BETWEEN NUMBER AND NUMBER
   nil
```

Examples of these declarations are:

```
set with size 100 of integer with range between 0 and 100 : x;
set of float : y; -- No hints in this one either!
set with size 10 of integer : z; -- the range is unknown
```

2.5. Expressions

Expressions in YASL resemble expressions in other "algorithmic" languages, including the set languages mentioned in the introduction. They are composed of operators (such as the well known mathematical operators) and operands. Although operators have precedence relations, parenthesises can be used to order the evaluation of computations.

2.5.1. Operands

Since operators operate on operands, it makes sense to discuss the operators of YASL first. The operands are the terminal symbols of the YASL grammar. There are two types of operands: scalar operands and set operands. The scalar operands are numbers and identifiers. Set operands are constructed by explicit set constant operators such as the set former or tuple former. The following are examples of both types of operands:

```
y                (variable)
5                (number)
```

```
( 1,2,3 )        (set former)
[ 2..5 ]         (tuple former)
```

2.5.2. Operators

Operators of YASL have a built-in precedence which is determined by the ordering of the operator symbol in the YASL grammar. Operators can be divided into four classes according to their semantics. These are:

2.5.2.1. Logical operators

The logical operators of YASL are the familiar logical operators: AND , OR and NOT. They are (respectively), logical disjunction, logical conjunction and logical negation.

2.5.2.2. Relational operators

The relational operators of YASL yield boolean results, and hence can be used with logical operators. The various symbols and their relations are as follows:

symbol	*relation*
MIN	minimum
MAX	maximum
<	less than
<=	less than or equal to
=	equal to
>=	greater than or equal to
>	greater than
<>	not equal
~=	not equal
SUBSET	is a subset of ...
IN	contained in ...

2.5.2.3. Arithmetic operators

Again, the arithmetic operators of YASL are identical to the operators in most algorithmic languages.

They are:

symbol	*operator*
+	Addition, Set union
-	Subtraction, Set intersection
*	Multiplication
/	Division

These operators are used on *numeric* types such as FLOAT and INTEGER. The set types have different meanings for these operators. These are detailed in the next section.

2.5.2.4. Set operators

As mentioned in the section above, the arithmetic operators take on a different meaning for set types. These are:

+,UNION	Set union
-	Set difference
*,INTERSECT	Set intersection
/	Symmetric set difference
WITH	Set addition
LESS	Set subtraction

Here are examples of all of the operators in action:

```
a < b
NOT c IN d
s - [ 1,2,3 ]
(a >= 10) AND (a < 20)
i IN { 1 .. 5 }
{ x SUCH THAT x IN s }
```

2.6. Statements

With the exception of the assignment and label statements, statements in YASL are used to alter the control flow. The syntax for statements is as follows:

```
realStatement ::=
   compoundStatement
   forStatement
   whileStatement
   forallStatement
```

```
existsStatement
ifStatement
assignmentStatement
labelStatement
```

2.6.1. Compound statements

Compound statements are a throwback to ALGOL-60; the BEGIN denotes the the beginning of a sequence of statements. The END denotes that the sequence has ended. Unlike ALGOL-60, BEGIN END pairs do not introduce a new lexical scoping environment. This was discussed in the section 3 of this chapter.

2.6.2. Assignment statements

Assignments are simple. The syntax is:

```
assignmentStatement ::= IDENTIFIER := expression
```

2.6.3. Labels

Labels were introduced into the syntax of YASL to name points in the program. These names were to be used in specifying timing requirements. A typical use would be to specify the time between two labels to be less than some performance requirement. The syntax is as follows:

```
labelStatement ::= [ IDENTIFIER ] statement
```

2.6.4. For statements

The YASL FOR statement is significantly different from the usual FOR statement. First, *the statement does not assume that one variable is going* to be set. Second, *multiple assign*ments are permitted in the body (normally a single limit and increment is proposed). The limit for the loop is a familiar boolean expression. The initial multiple assignments are done before entering the body of the loop (the initialization step). Next, the boolean expression is

evaluated. Next, the body of the loop is performed, followed by the next set of multiple assignments. Following these assignments, control returns to the top of the loop. The syntax is:

```
forStatement ::=
   FOR multipleAssignments THEN multipleAssignments
       forCondition booleanExpression DO statement
multipleAssignments ::=
   assignmentStatement multipleAssignmentTail
multipleAssignmentTail ::=
   , multipleAssignments
   nil
forCondition ::=
   WHILE
   UNTIL
```

Examples of this are found below:

```
FOR i := 0, j := 1 THEN i := i + 1 UNTIL i > 10 DO ...
FOR k := 0 THEN k := k + 2, k2 := k WHILE k < 100 DO ...
```

2.6.5. While statements

The while statement is identical to the ALGOL-60 while statement; the boolean expression is evaluated before entering the body of the loop; the body is evaluated and control returns to the boolean expression. The loop is exited when the boolean expression goes false. The syntax is:

```
whileStatement ::=
   WHILE booleanExpression DO statement
```

2.6.6. If statements

The IF statement is identical to most other IF statements (BCPL [RiW80] excepted). *Like the ALGOL-60 IF statement, this IF statement suffers from the "dangling else" syntactic* problem. This is resolved in favor of the nearest else. The exact syntax is:

```
ifStatement ::=
   IF booleanExpression THEN statement ifStatement*
```

```
ifStatement* ::=
   ELSE statement
   nil
```

2.6.7. Quantifiers

The set quantifiers resemble the **WHILE** loops in construction. The boolean condition in the **WHILE** statement corresponds with the test of set inclusion. The syntax of the quantifiers is:

```
forallStatement ::=
   FORALL IDENTIFIER IN setExpression DO statement
existsStatement ::=
   EXISTS IDENTIFIER IN setExpression DO statement
```

2.7. Miscellaneous

Each program must begin with a **PROGRAM** identifier. This is mainly to provide some identification of the program outside of comments. After declaring the program name, the program is constructed using statements. Statements come in two varieties: declarations and "real" statements, i.e., statements with actions. These were discussed in the previous sections.

```
program ::=
   PROGRAM IDENTIFIER statements END .
statements ::=
   statement statements*
statements* ::=
   ; statements
   nil
statement ::=
   declaration
   realStatement
```

3. Syntax (BNF)

This is the complete grammar for YASL:

```
program ::=
   PROGRAM IDENTIFIER statements END .
```



```
statements ::=
    statement statements*
statements* ::=
    ; statements
    nil
statement ::=
    declaration
    realStatement
declaration ::=
    setDeclaration
    tupleDeclaration
    integerDeclaration
    floatDeclaration
    characterDeclaration
realStatement ::=
    compoundStatement
    forStatement
    whileStatement
    forallStatement
    existsStatement
    ifStatement
    assignmentStatement
    labelStatement
setDeclaration ::=
    SET optionalSize OF declaration
tupleDeclaration ::=
    TUPLE optionalSize OF declaration
integerDeclaration ::=
    INTEGER optionalRange : idList
floatDeclaration ::=
    FLOAT optionalRange : idList
characterDeclaration ::=
    CHARACTER : idList
optionalSize ::=
    WITH SIZE NUMBER
    nil
optionalRange ::=
    WITH RANGE BETWEEN NUMBER AND NUMBER
    nil
compoundStatement ::=
    BEGIN statements END
assignmentStatement ::=
    IDENTIFIER := expression
labelStatement ::=
    [ IDENTIFIER ] statement
forStatement ::=
    FOR multipleAssignments THEN multipleAssignments forCondition booleanExpression DO statement
multipleAssignments ::=
    assignmentStatement multipleAssignmentTail
multipleAssignmentTail ::=
    , multipleAssignments
    nil
forCondition ::=
    WHILE
    UNTIL
whileStatement ::=
    WHILE booleanExpression DO statement
forallStatement ::=
    FORALL forallStatementVariable
forallStatementVariable ::=
    IDENTIFIER IN setExpression forallStatementQualifier forallStatementBody
forallStatementQualifier ::=
    SUCH THAT booleanExpression
    nil
```

forallStatementBody ::=
    DO statement
existsStatement ::=
    FORALL existsStatementVariable
existsStatementVariable ::=
    IDENTIFIER IN setExpression existsStatementQualifier existsStatementBody
existsStatementQualifier ::=
    SUCH THAT booleanExpression
    nil
existsStatementBody ::=
    DO statement
ifStatement ::=
    IF booleanExpression THEN statement ifStatement*
ifStatement* ::=
    ELSE statement
    nil
booleanExpression ::=
    expression
expression ::=
    disjunction
disjunction ::=
    conjunction disjunction*
disjunction* ::=
    OR disjunction
    nil
conjunction ::=
    not conjunction*
conjunction* ::=
    AND conjunction
    nil
not ::=
    NOT not*
    relationalExpression
not* ::=
    relationalExpression
relationalExpression ::=
    boundedExpression relationalExpression*
relationalExpression* ::=
    relationalOp boundedExpression
    nil
boundedExpression ::=
    addition boundedExpression*
boundedExpression* ::=
    boundingOp boundedExpression
    nil
boundingOp ::=
    MAX
    MIN
relationalOp ::=
    <
    <=
    <>
    >=
    >
    =
    '=
    SUBSET
    IN
addition ::=
    multiplication addition*
addition* ::=
    additionOp addition
    nil
additionOp ::=

    +
    -
    WITH
    LESS
multiplication ::=
    factor multiplication*
multiplication* ::=
    multiplicationOp multiplication
    nil
multiplicationOp ::=
    *
    /
    MOD
    DIV
factor ::=
    - factor*
    primary
factor* ::=
    primary
primary ::=
    NUMBER
    # setSize
    setExpression
setSize ::=
    setExpression
setExpression ::=
    setTerm
setTerm ::=
    setFactor setTerm*
setTerm* ::=
    UNION setTerm
    nil
setFactor ::=
    setPrimary setFactor*
setFactor* ::=
    INTERSECT setFactor
    nil
setPrimary ::=
    emptySet
    IDENTIFIER
    (expression)
    { setFormer }
    { tupleFormer }
emptySet ::=
    PHI
setFormer ::=
    IDENTIFIER IN setExpression SUCH THAT booleanExpression
    setConstantList
setConstantList ::=
    constant setConstantLists
setConstantLists ::=
    . . setConstantRange
    setConstantList*
setConstantList* ::=
    , setConstantList
    nil
setConstantRange ::=
    constant setConstantList*
tupleFormer ::=
    IDENTIFIER IN setExpression SUCH THAT booleanExpression
    tupleConstantList
tupleConstantList ::=
    constant tupleConstantLists
tupleConstantLists ::=

```
    . . tupleConstantRange
    tupleConstantList*
tupleConstantList* ::=
    , tupleConstantList
    nil
tupleConstantRange ::=
    constant tupleConstantList*
constant ::=
    NUMBER
idList ::=
    IDENTIFIER idList*
idList* ::=
    , idList
    nil
```


Pages 115 to 154 were omitted from the technical report edition due to cost considerations. Copies of these pages can be obtained either from the Xerox University Microfilm edition or directly from the author.

"Your intuition is excellent but another viewpoint could be helpful"

Appendix D

Digital signal processing Languages

1. Introduction

Digital signal processing is an interesting domain for both hardware and language design. This is due, in part, to the different properties of digital signal processing algorithms. In particular, digital signal processing algorithms have some of the following features:

- performance criteria

  There are many types of performance criteria for digital signal processing. For example, filters have bands, Q, noise limits and other specifications. Transforms also have limits on performance, particularly speed and time limits. These are useful criteria for automatic selection.

- applicative nature

  An "applicative nature" is a loose term that means (in this context) a lack of side effects and the ability to cascade (pipeline) functions. Also, this means that the algorithms are *not* tied to state transitions.

- parallel functions

  Digital signal processing algorithms often have functions that can be performed in parallel. This is particularly true since (as stated above) many algorithms lack side effects.

- varying arithmetic formats

  Digital signal processing often involves the use of different arithmetic representations due to differing demands for precision, noise and other performance parameters.

There are several different levels to look at digital signal processing. One level is at the level of individual samples (from an analog to digital converter). This can be called the *signal* level. Examples of signal level processing are the computation of filters, transforms and similar direct data manipulation. Another higher level is concerned with algorithms; how to fit the functions at the signal level together to perform a task. This can be called the system level.

The history of languages for digital signal processing dates back to earlier days of computing. BLODI [Kar65], for example, was an early block diagram compiler. Unfortunately, such languages lack the power to handle such algorithms as the Fast Fourier Transform.

Recently, there has been work on applying data abstraction and typing mechanisms to digital signal processing. Gethöffer [Get80] gives a sketch of a language (SIPROL) that has some data abstraction capability in it. (A PASCAL-like language with a few added types). Kopec's thesis [Kop80] is a much more complete description of how CLU [LAM81] can be extended through judicious use of data abstraction to cover both the signal and system aspects of digital signal processing.

However, both Kopec and Gethöffer deal with types of a high level but not at a very high level. A very high level specification does not deal with the sampled data but rather at the level of connecting functions without knowing the underlying implementation.

2. A description of CLASP

CLASP[1] was designed to handle the module to module level of description of digital signal processing functions. Here, the modules perform fairly high level functions such as

[1] *Complex Language for Attacking Signal Processing.*

filtering and transforms, not low level functions such as registers (delay).

Much of the syntactic structure for the language was borrowed from the set language (YASL) used in most of the thesis. In particular, many of the control structures are similar (if not identical). Naturally, there are changes to the data types and some of the looping structures.

2.1. Features unique to CLASP

Although signal processing algorithms are varied, there are two basic structures: filters and transforms. The next two sections consider how to express filtering and transform operations.

2.1.1. Filters

Consider the specification of a filter. A filter has several performance criteria. Among these criteria are bandwidth, quality (Q), sideband noise, roundoff error and noise. So that CLASP is an effective digital signal processing specification language, it should be able to express these parameters as part of the program. Another approach might be to attach assertions. CLASP takes the view that such assertions should be visible and are part of the specification; as much as any other property in the set domain of YASL. Filters can either be declared (in effect becoming a function) or used directly within an expression. Naturally, a declared filter has constant upper and lower bounds. So, lowpass filter from DC to A (440 hertz) could be declared as

```
declare filter from DC to 440 : Afilter;
a := Afilter(input);
```

or it could just be used in an expression:

```
a := filter input from DC to 440 : aFilterTwo
```


Of course, specifications can be added to the filter just as set declarations can be added in YASL. Consider the following sample declaration:

```
filter input from f0-1000 to f0+1000
with passband ripple 5 db and
with stopband attenuation 66 db down;
```

The complete syntax for a filter is as follows:

```
filterDeclaration ::=
   FILTER FROM constant TO constant filterSpecs : IDENTIFIER
filteredExpression ::=
   FILTER expression FROM expression TO expression filterEnd
filterEnd ::=
   filterSpecs filterName
filterSpecs ::=
   WITH filterSpec
   nil
filterSpec* ::=
   AND filterSpecs
   nil
filterSpec ::=
   Q OF constant filterSpec*
   PASSBAND RIPPLE OF constant DB filterSpec*
   STOPBAND ATTENUATION OF constant DB measureSign filterSpec*
measureSign ::=
   DOWN
   nil
filterName ::=
   : IDENTIFIER
   nil
```

Note that the filter specifications are optional but should be specified by the user if the proper filter is to be selected.

2.1.2. Transforms

Now, consider the use of a transform (Fourier, Laplace, Hilbert, etc.). One such example occurs in digital mixing where convolution of two signals is a common operation. This can expressed in a signal processing language by transforming the two input sequences (tuples) into the frequency domain and performing the convolution. This is written in CLASP as:

```
transform input-seq from time domain into frequency domain;
transform envelope-seq from time domain into frequency domain;
transform convolve(input-seq, envelope-seq) into time domain;
```

Note that the Fourier transform needs complex numbers, therefore complex numbers must be included in the primitive types of CLASP. Note also that the Discrete Fourier Transform (DFT) may have implementations in silicon that take less time and area than the Fast Fourier transform (FFT) [FoK79]. Of course, the choice of implementation is done automatically by the selection phase of the compiling system.

In retrospect, a better design would have been to introduce separate types for each transform domain. Then, the transform would become a coercion operation between types. Unfortunately, the demonstration system's simple system of type declarations wouldn't be able to handle these types. This is, however, strictly an implementation issue.

The syntax of a transform expressions is:

```
transformExpression ::=
   TRANSFORM setExpression FROM transformDomain DOMAIN
                           TO transformDomain DOMAIN
transformDomain ::=
   TIME
   FREQUENCY
```

2.1.3. Special iterative forms

The EVERY statement is designed to express the fundamental relationship between sampling rate and the length of the microprogram. In particular, the EVERY statement is used in the outer loop of the program so that the sampling rate can be specified. The sampling rate is the reciprocal of the time specified by the "timeMeasurement" part of the statement. Unfortunately, the implementation of the EVERY statement is not complete. This is because the present SILI system doesn't perform a detailed timing analysis, therefore, it is incapable of calculating the length of the wait state required at end of the loop. Note that wait nodes (in the control flow graph) are needed when the microprogram is too short. Microcode compaction is needed when the microprogram is too long!


```
everyStatement ::=
   EVERY timeMeasurement DO statement
timeMeasurement ::=
   expression timeUnits
timeUnits ::=
   MILLISECONDS
   MS
   MICROSECONDS
   UB
```

2.1.4. Functions

Most of the functions used in a digital signal processing setting are expressed in CLASP as functions. This function calling style is converted by the compiler into the proper staging of functional units. This conversion will be discussed in the next section.

3. Generation of machines from CLASP specifications

3.1. Introduction

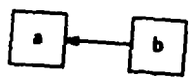
The next question is: given a CLASP program, how does the compiler convert it to a machine? The answer lies in the control and data flow analysis routines. Since the flow analysis routines are table driven, the flow analysis for CLASP can (and does) differ from that of YASL. But the main question remains: how to translate the assignments and function calls into the appropriate connection of modules.

3.1.1. Functions

For example, the CLASP code

```
a := b
```

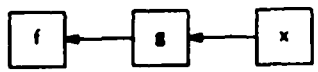
is translated into the following connection of modules



in exactly the same way that it was implemented for YASL. The CLASP code fragment

f(g(x))

is translated into these modules



using a very similar technique.

3.2. From specifications to types

As the section on filters noted, there is a syntactic method for specifying the properties of the filter. This can be used to insert the appropriate values into the type declarations, much as YASL did. For example, the filter specification in section 2.1.1 can be represented in the type block for the filter as `((passband-ripple 5) (stopband-attenuation 66))`.

3.3. Metrics

The metrics for the digital signal processing are different from the metrics of set languages. In particular, emphasis is placed on the ability to use pipelined components. Therefore, a good metric to use is ATP, where P is the period of the device. Cappello and Steiglitz [CaS81] used this metric in their work on analyzing pipelined serial digital signal processing circuits. The SILI implementation uses the same metric as YASL, if only because of the convenience (and the lack of a good periodicity calculation).

3.4. Calculation of coefficients

Another task that SILI doesn't perform is the calculation of filter coefficients. This isn't to say that this subtask isn't within SILI's realm of expertise – rather that it was omitted for reasons of time and necessity. However, this brings up an interesting point. Consider the case of the example program – here a filter is defined in terms of a center frequency with a constant width. Now, the matcher selects a variable filter (as it should) – but should SILI generate a machine that calculates the coefficients of this filter at "runtime" (i.e., during filter operation) or during compile time? The answer should be obvious – certain filter coefficient computations are quite extensive, therefore most, if not *all* coefficient calculations should be done at compile time. But how? In the case of the example program, the center frequencies are stored in a tuple and accessed sequentially from that tuple. The compiler system should be capable of realizing that these frequencies (in the tuple) are constants, and furthermore, that the tuple can be changed from a tuple of frequencies to a tuple of coefficients. This is an extremely important "optimization" and should be part of any compiler for a very high level signal processing language.

3.5. Architecture and Microcode generation

3.5.1. Architecture

Because of the looping structures, the default architectures (i.e., before critics) have *multiplexed hardware*. For example, the implementation of the digital Touch-Tone decoder (see the last section) has two loops for the four filter bands. Therefore, only one filter chain is created and is shared over the four bands. If the user needs all four filters implemented for speed reasons, this can be specified by an appropriate time demand. The "out of time" critic must be prepared to "unroll" the loop in much the same fashion as a conventional compiler (see [AhU77], pp. 471-472).

3.5.1.1. Word length effects

While word length in the machine is dependent on the program (and the eventual application), there are other side effects. In particular, word length can effect overflow and round-off, which in turn can effect the noise figures [MuR76].

3.5.1.2. Parallel v.s. Serial architectures

Like YASL, CLASP doesn't make a committment to any to any form of parallel or serial architecture. Interest in serial architectures for signal processing has been increasing since Lyon's paper [Lyo81] [Lyo80]. He draws his inspiration from an earlier paper (now 15 years old) of Jackson, Kaiser and McDonald [JKM68]. Lyon extended their work by introducing interface standards between modules and also using hierarchy in designs. Lyon's serial "philosophy" as seized by a group at Edinburgh [Den] and used in a simple silicon compiler (FIRST) [Ber81]. FIRST uses a fairly fixed placement scheme and a small number of predefined operators. The FIRST language itself is a rather simple and low level register transfer language. This contrasts sharply with CLASP, which ignores placement and routing issues and uses types of very high level. However, note that a tool like FIRST could be used to generate the lower level cells for CLASP.

3.5.2. Microcode generation

Because the machine generation phase is intimately tied to the notion of "data flow = data path, control flow = control store", the machine generation of CLASP programs does not differ from that of the YASL set domain. So, control constructs are translated into the control flow graph and the graph is used to generate the microcode store. The use of microcode for signal processing machines is not new of course. Allen [All75] has a review of some microcoded digital signal processing machines circa 1975. However, there is one "feature" that should be noted: For real time machines, the sampling rate is proportional to the length of the microstore; specifically, the

sampling rate = 1/(length of the microstore * speed of the slowest step)

Of course, should the machine have a small microstore, then wait states must be used. If the microstore is too large (more often the case), then steps must be taken to compact it. This is either the job of microstore compaction (discussed in greater detail in chapter 7) or the

critics, which can possibly reduce the speed of the slowest step.

4. An example

Using the touch tone decoder example, the following block diagram will be generated by SILI:

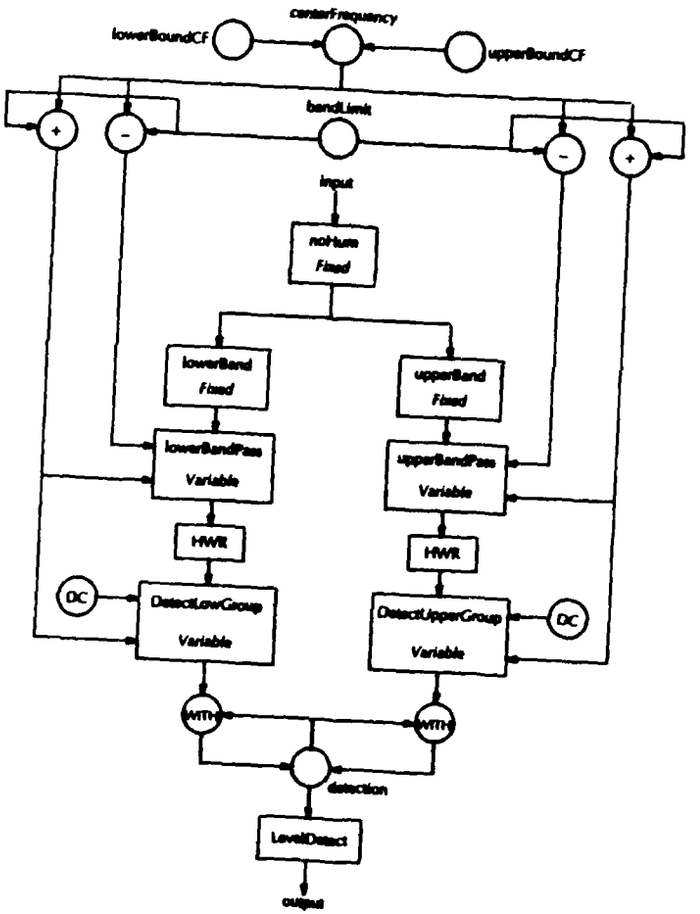


Figure D.1 Touch tone decoder data flow graph

5. Conclusion

The CLASP language is significantly different from past digital signal processing languages. It permits users who are naïve in the design of digital signal processing circuits to specify a design that can be automatically constructed. Like the set language YASL, CLASP depends on an extensive library built using other tools. CLASP also allows the users to treat digital signal processing functions as "black boxes" and ignore the underlying implementations. Of course this is the aim of Very High Level Languages such as YASL and CLASP!

6. Syntax (BNF)

This is the complete grammar for CLASP:

```
program ::=
    MODULE IDENTIFIER statements END .
statements ::=
    statement statements*
statements* ::=
    ; statements
    nil
statement ::=
    declaration
    realStatement
declaration ::=
    DECLARE typeDeclaration
typeDeclaration ::=
    complexDeclaration
    numericTypes
    setDeclaration
    tupleDeclaration
    filterDeclaration
    transformDeclaration
complexDeclaration ::=
    COMPLEX OF numericTypes
numericTypes ::=
    floatDeclaration
    integerDeclaration
realStatement ::=
    compoundStatement
    forStatement
    whileStatement
    forallStatement
    existsStatement
    everyStatement
    ifStatement
    testStatement
    identifierStart
    labelStatement
setDeclaration ::=
    SET optionalSize OF typeDeclaration
tupleDeclaration ::=
    TUPLE optionalSize OF typeDeclaration
```



```
integerDeclaration ::=
    INTEGER optionalRange : idList
floatDeclaration ::=
    FLOAT optionalRange : idList
filterDeclaration ::=
    FILTER FROM constant TO constant filterSpecs : IDENTIFIER
transformDeclaration ::=
    TRANSFORM FROM transformDomain DOMAIN TO transformDomain DOMAIN : IDENTIFIER
transformDomain ::=
    TIME
    FREQUENCY
optionalSize ::=
    WITH SIZE NUMBER
    nil
optionalRange ::=
    WITH RANGE BETWEEN NUMBER AND NUMBER
    nil
optionalSize ::=
    WITH SIZE NUMBER
    nil
optionalRange ::=
    WITH RANGE BETWEEN NUMBER AND NUMBER
    nil
compoundStatement ::=
    BEGIN statements END
identifierStart ::=
    IDENTIFIER identifierTail
identifierTail ::=
    assignmentStatement
    callProcedure
assignmentStatement ::=
    := expression
callProcedure ::=
    ( proceduresCall
labelStatement ::=
    | IDENTIFIER | statement
forStatement ::=
    FOR multipleAssignments THEN multipleAssignments forCondition booleanExpression DO statement
multipleAssignments ::=
    assignmentStatement multipleAssignmentTail
multipleAssignmentTail ::=
    , multipleAssignments
    nil
forCondition ::=
    WHILE
    UNTIL
whileStatement ::=
    WHILE booleanExpression DO statement
forallStatement ::=
    FOREACH forallStatementVariable
forallStatementVariable ::=
    IDENTIFIER IN setExpression forallStatementQualifier forallStatementBody
forallStatementQualifier ::=
    SUCH THAT booleanExpression
    nil
forallStatementBody ::=
    DO statement
existsStatement ::=
    FORALL existsStatementVariable
existsStatementVariable ::=
    IDENTIFIER IN setExpression existsStatementQualifier existsStatementBody
existsStatementQualifier ::=
    SUCH THAT booleanExpression
    nil
```

```
existsStatementBody ::=
    DO statement
everyStatement ::=
    EVERY timeMeasurement DO statement
timeMeasurement ::=
    constant timeUnits
timeUnits ::=
    MILLISECONDS
    MS
    MICROSECONDS
    US
ifStatement ::=
    IF booleanExpression THEN statement
testStatement ::=
    TEST booleanExpression IFSO statement IFNOT statement
booleanExpression ::=
    expression
expression ::=
    disjunction
disjunction ::=
    conjunction disjunction*
disjunction* ::=
    OR disjunction
    nil
conjunction ::=
    not conjunction*
conjunction* ::=
    AND conjunction
    nil
not ::=
    NOT not*
    relationalExpression
not* ::=
    relationalExpression
relationalExpression ::=
    boundedExpression relationalExpression*
relationalExpression* ::=
    relationalOp boundedExpression
    nil
boundedExpression ::=
    addition boundedExpression*
boundedExpression* ::=
    boundingOp boundedExpression
    nil
boundingOp ::=
    MAX
    MIN
relationalOp ::=
    <
    <=
    <>
    >=
    >
    =
    ~=
    SUBSET
    IN
addition ::=
    multiplication addition*
addition* ::=
    additionOp addition
    nil
additionOp ::=
    +
```



```
    -
    PLUS
    MINUS
multiplication ::=
    factor multiplication*
multiplication* ::=
    multiplicationOp multiplication
    nil
multiplicationOp ::=
    *
    /
factor ::=
    - factor*
    primary
factor* ::=
    primary
primary ::=
    NUMBER
    infinityConstant
    dcConstant
    # setExpression
    filteredExpression
    setExpression
infinityConstant ::=
    INFINITY
dcConstant ::=
    DC
setExpression ::=
    setTerm
filteredExpression ::=
    FILTER expression FROM expression TO expression filterSpecs filterName
filterSpecs ::=
    WITH filterSpec
    nil
filterSpec* ::=
    AND filterSpecs
    nil
filterSpec ::=
    Q OF constant filterSpec*
    PASSBAND RIPPLE OF constant DB filterSpec*
    STOPBAND ATTENUATION OF constant DB measureDirection filterSpec*
measureDirection ::=
    DOWN
    nil
filterName ::=
    : IDENTIFIER
    nil
setTerm ::=
    setFactor setTerm*
setTerm* ::=
    UNION setTerm
    nil
setFactor ::=
    setPrimary setFactor*
setFactor* ::=
    INTERSECT setFactor
    nil
setPrimary ::=
    emptySet
    id
    ( expression )
    setTupleFormers
setTupleFormers ::=
    { setFormer
```

```
    ( tupleFormer
emptySet ::=
    PHI
id ::=
    IDENTIFIER optionalCall
optionalCall ::=
    ( procedureCall
    nil
procedureCall ::=
    expression )
setFormer ::=
    IDENTIFIER IN setExpression SUCH THAT booleanExpression
    setConstantList
setConstantList ::=
    constant setConstantLists
setConstantLists ::=
    . . setConstantRange
    setConstantList*
setConstantList* ::=
    , setConstantList
    nil
setConstantRange ::=
    constant setConstantList*
tupleFormer ::=
    IDENTIFIER IN setExpression SUCH THAT booleanExpression
    tupleConstantList
tupleConstantList ::=
    constant tupleConstantLists
tupleConstantLists ::=
    . . tupleConstantRange
    tupleConstantList*
tupleConstantList* ::=
    , tupleConstantList
    tupleConstantListEnd
tupleConstantListEnd ::=
    ]
tupleConstantRange ::=
    constant tupleConstantList*
constantList ::=
    constant constantLists
    nil
constantLists ::=
    . . constant
    constantList*
constantList* ::=
    , constantList
    nil
constant ::=
    NUMBER
    InfinityConstant
idList ::=
    IDENTIFIER idList*
idList* ::=
    , idList
    nil
```

170-235

Pages 170 to 235 were omitted from the technical report edition due to cost considerations. Copies of these pages can be obtained either from the Xerox University Microfilm edition or directly from the author.

"Rely on yourself if you want to get things done"

Appendix E

Programming Vignettes

1. Introduction

This program was by far the largest Lisp program the author had ever written. In creating such a program, the author fought and won (and lost) several battles. Some of these are recorded below as well as reflections on *Lisp as a tool for building experimental systems.*

2. Global name space problems

As any Lisp user knows, there is one space for Lisp names: the oblist. Therefore, the author was very careful to prefix each function (and atom) name with the name of the file that the function was written to. Free variables were kept to a minimum and the use of lambda variables was maximized. Dynamic scoping was used occasionally, but it makes the program harder to read from just the code. The ounce of prevention (unique names) worked; not one bug was due to conflicting names. Recent Lisp systems have "packages" that permit the user to keep these in separate address spaces. Of course, there was only one programmer, so it was much easier to avoid conflicts.

3. Fighting with the Lisp implementation

The particular dialect of Lisp that was used for the system (called "Franz Lisp") was

clearly not designed with large system building in mind.[1]

Pushdown list overflow can result in a global reset – leaving the user (author) bewildered (and angry) as all the stack state was gone! There were no facilities for doing the kind of interrogation that SCOPE [Mas80] can do. As a result, it was often necessary to prettyprint one function after another, all the way down the calling hierarchy.

The author will claim here that solid facilities such as tracing, breaking and SCOPE-like interrogation facilities are a necessity in any language when building a large system. Yet, very few systems have these facilities, especially the so called "algorithmic" languages.

4. Reflections on using Lisp

The author's choice of Lisp as a system building language was motivated by several concerns. First, Lisp removes worries about faulty memory allocation and pointer chasing. *This turned out to be a true plus. Second, the author wanted the ability to radically change* SIL if a horrendous difficulty was uncovered. This happened at least once and the system was "torn apart" and reconstructed in the period of about a day and a half. This was also a plus. Although speed was not an ultimate concern, the speed of the interpreter was often unbearable during debugging. This was definitely a minus.

Programming in Lisp is definitely different. Sandewall [San78] has an interesting article about Lisp and Lisp programming systems. One of the more interesting facets of programming in Lisp is building the system from bottom up and from top down simultaneously.

At the advice of James Allen, the author avoided using so called "hairy data structures" by using GENSYM symbols instead of creating pointers with CONS. While this introduced additional complexity (the need to eval the GENSYM names to get the values), it did reduce the complexity of pointer management.

---

[1] In all fairness, Franz was designed to port a rather large Maclisp program, Macsyma, not to build new systems.

5. A tale of two systems

There were two experimental systems constructed over the course of a year and a half or more. The second system turned out to be an almost total rewrite of the first system. Following the advice of the thesis committee, the author's first system had *no* facilities for program analysis. Information about the programs was generated by hand by the author and given to the program. This proved to be tedious and error prone. There was also some question about what exactly was required by the compiler. All of these problems were remedied in the second implementation.

The first system also used a different constraint propagation algorithm. In particular, constraints were propagated across the data flow graph until either a constraint conflict occurred (like a serial path met a parallel path) or the constraint was already present in a node (i.e., propagating the property "serial" into a node already marked "serial".). This is clearly wrong because the constraints of one selection should not condemn the rest of the selections to be constrained unless they are all connected. This ultimately goes back to the notion that constraints represent compatibility between ports. The second system only propagates constraints across one arc (to the connected node) in the data flow graph.

The organization SILI has changed radically since the first implementation of SILI. In particular, the strategy for attaching implementations in the library to nodes in the data flow graph has become the most complex part of the program.

Initially (in the first implementation), this was done by a straight forward table lookup. However, this also depended on the operators being part of the definition of the identifiers. This is clearly an unrealistic model. Eventually, a primitive matcher was written. This matcher suffered from the following problem: When a match was made between the library description and the data flow graph, a single record was created (called an instance). Unfortunately, this doesn't account for modules with differing control signals that produce different

functions. The next version of the matcher created match records that in turn were tied to a single instance of the library module. This proved not only to be a reasonable strategy but also had intuitive appeal.

In the second implementation, data flow and control flow analysis routines started out being entirely separate. This arrangement worked fine until it became time to generate the microcode. At that time it became obvious that each data flow node needed a list of control flow nodes that "used" the data flow node. Fortunately, the fix was easy. The control flow grammar was changed to have a construct that transferred control to the data flow analysis routines. The current control flow node was then inserted in any data flow nodes subsequently generated.

In order to generate estimates for the control store (and the jump multiplexer), it was convenient to generate a flow node. This is desirable because it permits all the standard mechanisms of binding, metrics and critics to be applied to the new selection. The generation of these new nodes turns out to be relatively straight forward. First, a special data flow node is created with a property list that can be bound by the binding procedure using the library's representation for the implementation. Next, a match node is created (without calling the matcher) establishing a pseudo-match between the implementation and the new data flow node. The next to last step is the creation of an instance that ties the match to the library implementation. The final step is to call the binder, which proceeds smoothly from this point on.

6. The implementation of critics

Halfway through the implementation of the second system, it became apparent that contexts would be required to implement the critics. The reason is as follows: since the critics change the data flow graph, there must be a mechanism to change the data flow graph without effecting the other nodes (states) in the search tree. A context system like that used

in Conniver [McS72] would have been just right here.

7. Debugging the library

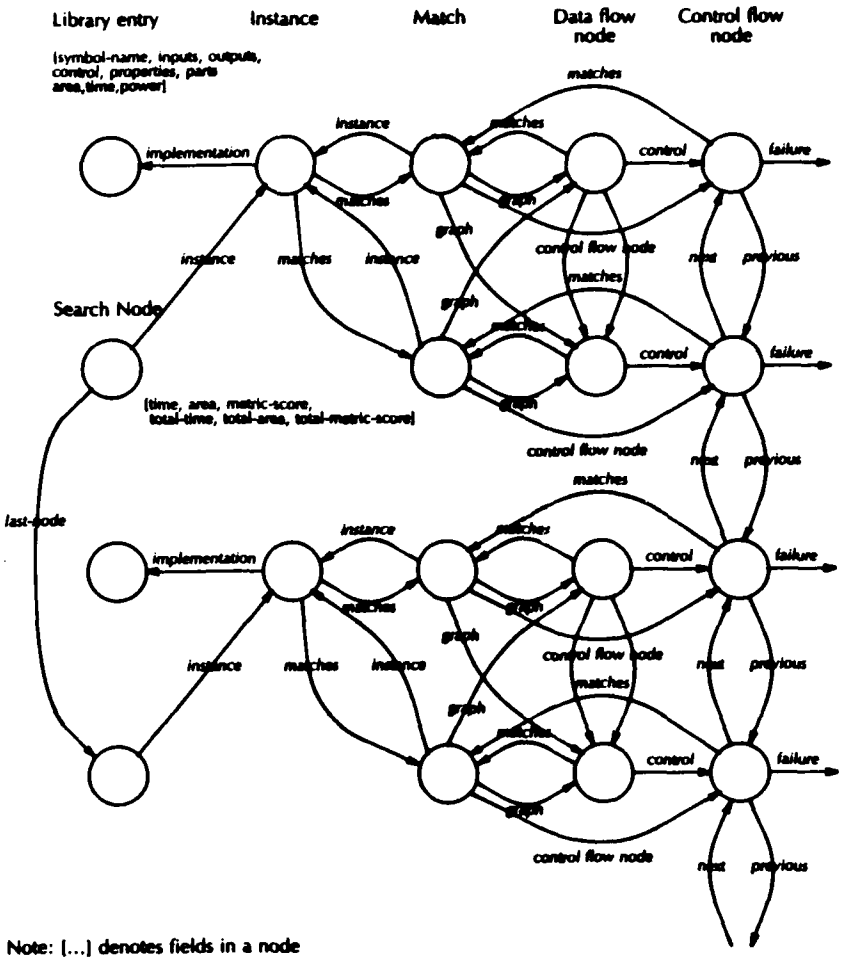
Unfortunately, debugging the library specification turned out to be an extremely error prone and time consuming operation. Misspellings and improper graph descriptions would often not appear until late in the session. Since the program can take hours to run (interpretively), this was literally a waste of time. The solution would be to write a specification checker. Such a checker should look for undeclared names (and functions). Such a system would save *days of debugging time.*

8. Specification of declarations

Perhaps the most language dependent part of Sru was the specification for the declarations. Declarations were done by walking the parse tree left to right and calling a function for specific nodes. Naturally, all the semantics of the declarations resided in the functions. And since the functions were written in LISP, it was easy to do anything desired. Note also that the left to right traversal made declarations of the form "declare type: identifier-list" more advantageous that the "declare identifier-list : type", since it was easier to write a function *that stuck types on one id at a time.*

9. Hairy data structures

As an instructive measure, the next page illustrates the connection between the data flow nodes, match nodes, control flow nodes, instances of library modules and the library module definitions.



Note: [...] denotes fields in a node

Figure E.1 Hairy data structures

Bibliography

[AcD79] W. B. Ackerman and J. B. Dennis.
VAL - A value-oriented algorithmic language; Preliminary reference manual.
TR-218, MIT, June 1979.

[Ack82] W. B. Ackerman.
Data flow languages.
Computer, 15(2):15-25, February 1982.

[Age76] T. Agerwala.
Microprogram Optimization: A Survey.
IEEE Transactions on Computers, C-25(10):962-973, October 1976.

[AhJ76] A. V. Aho and S. C. Johnson.
Optimal Code Generation for Expression Trees.
Journal of the ACM, 23(3):488-501, 1976.

[AhU77] A. V. Aho and J. D. Ullman.
Principles of Compiler Design.
Addison Wesley, Reading, MA, 1977.

[All70] F. E. Allen.
Control Flow Analysis.
SIGPLAN Notices, 5(7):1-19, July 1970.

[All75] J. Allen.
Computer architecture for signal processing.
Proceedings of the IEEE, 63(4):624-633, April 1975.

[Arv79] Arvind.
Decomposing a program for multiple processors.
Proc. of International Conference on Parallel Processing, pages 7-14, 1979.

[Bac78] J. Backus.
Can programming be liberated from the von Neumann style?.
Communications of the ACM, 21(8):613-641, August 1978.

[Bar78] M. R. Barbacci.
An Introduction to ISPS.
Technical Report 78-137, Carnegie-Mellon Univ., August 1978.

[Bar81] M. R. Barbacci.
Instruction Set Processor Specifications (ISPS): The Notation and its Applications.
IEEE Transactions on Computers, C-30(1):24-40, January 1981.
(also Carnegie-Mellon Univ. Technical Report 79-123)

[BaH80] J. Batali and A. Hartheimer.
The Design Procedure Language Manual.
VLSI Memo 80-31, MIT, September 1980.

[Bat81] J. Batali.
An Introduction to DPL.
VLSI Memo 81-65, MIT, October 1981.

[BMS81] J. Batali, N. Mayle and H. Shrobe.
The DPL/Daedalus Design Environment.
pp. 183-192 in *VLSI-81*.
Academic Press, New York, NY, 1981.

[Bau81] G. Baudet.
On the area required for VLSI circuits.
pp. 100-107 in *Carnegie-Mellon Univ. Conference on VLSI Systems and Computations*, ed. H. T. Kung, B. Sproull and G. Steele.
Computer Science Press, Rockville, MD, 1981.

[Ber81] N. Bergmann.
A Case Study of the FIRST Silicon Compiler.
pp. 413-430 in *VLSI-81*.
Academic Press, New York, NY, 1981.

[BoJ66] C. Bohm and Jacobini.
Flow diagrams, Turing Machines and Languages with only two formation rules.
Communications of the ACM, 9(5):366-371, May 1966.

[Bro76] A. Brown.
Qualitative Knowledge, Causal Reasoning and the Localization of Failures.
AI-TR-362, PhD thesis, MIT, November 1976.

[Bur82] G. R. Burke.
Control Schemes for VLSI Microprocessors.
MICRO-16, pages 91-95, 1982.

[CaS81] P. Cappello and K. Steiglitz.
Digital Signal Processing applications of systolic algorithms.
pp. 245-254 in *Carnegie-Mellon Univ. Conference on VLSI Systems and Computations*, ed. H. T. Kung, B. Sproull and G. Steele.
Computer Science Press, Rockville, MD, 1981.

[Cat78] R. G. G. Cattell.
Formalization and Automatic Derivation of Code Genrators.
Technical Report 78-115, PhD thesis, Carnegie-Mellon Univ., April 1978.
(Also published by UMI Press).

[Cat79] R. G. G. Cattell.
Code Generation and Machine Descriptions.
CSL 79-8, Xerox PARC, October 1979.

[Cat80] R. G. G. Cattell.
Automatic Derivation of Code Generators from Machine Descriptions.
ACM Transactions on Programming Languages and Systems, 2(2):173-190, April 1980.

[CAC81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markstein.
Register Allocation via Coloring.

Computer Languages, 6(1):47-58, 1981.

[ChM81] B. Chazelle and L. Monier.
A Model of Computation for VLSI with Related Complexity Results.
Technical Report 81-107, Carnegie-Mellon Univ., February 1981.

[CoZ] J. Cohen and C. Zuckerman.
Two languages for Estimating Program Efficiency.
Communications of the ACM, 17(6):301-307.

[Das80] S. Dasgupta.
Some Aspects of High Level Microprogramming.
ACM Computing Surveys, pages 295-324, September 1980.

[DLS81] S. Davidson, D. Landskov, B. D. Shriver and P. W. Mallett.
Some experiments in Local Microcode Compaction for Horizontal Machines.
IEEE Transactions on Computers, pages 460-477, July 1981.

[deS80] J. deKleer and G. J. Sussman.
Propagation of constraints applied to circuit synthesis.
Circuit Theory and Applications, 8:127-144, 1980.

[deK] J. deKleer.
A theory of plans for electronic circuits.
MIT AI Working Paper 144.

[Den79] J. Dennis.
Varieties of data flow computers.
First Int. Conf. on Dist. Computing, pages 430-439, 1979.

[Den] P. B. Denyer.
An introduction to Bit-serial Architectures for VLSI Signal Processing.
pp. 225-241 in *VLSI Architecture*, ed. B. Randell and P. C. Treleaven.
Prentice Hall, Englewood Cliffs, NJ.

[DeS79] R. B. K. Dewar and E. Schonberg.
The Elements of SETL style.
Proceedings of the ACM National Conference, pages 24-32, 1979.

[DGL79] R. B. K. Dewar, A. Grand, S-C Liu, J. T. Schwartz and E. Schonberg.
Program by refinement, as exemplified by the SETL Representation sublanguage.
ACM Transactions on Programming Languages and Systems, 1(1):27-49, July 1979.

[DPS] S. W. Director, A. C. Parker,, D. P. Siewiorek and D. E. Thomas.
A Design Methodology and Computer Aids for Digital VLSI Systems.
Carnegie-Mellon Univ. Computer Science Review 1980-1981.

[Don81] V. Donzeau-Gouge.
Denotational definition of properties of program correctness.
pp. 343-379 in *Program Flow Analysis*, ed. S. S. Muchnick and N. D. Jones.
Prentice Hall, Englewood Cliffs, NJ, 1981.

[Dor] J. Doran.
An approach to automatic problem-solving.
pp. 105-123 in *Machine Intelligence 1*.
Edinburgh University Press.

[Ear73] J. Earley.
Relational Level Data Structures for Programming Languages.
Acta Informatica, 2:293-309, 1973.

[Ear74] J. Earley.
High level operations in automatic programming.
SIGPLAN Notices, pages 34-42, 1974.
(also UC Berkeley Technical Report)

[FaR78] D. G. Fairbairn and J. H. Rowson.
ICARUS - An interactive IC Layout Program.
15th Design Automation Conf., pages 188-192, 1978.

[FKZ76] R. Farrow, K. Kennedy and L. Zucconi.
Graph Grammars and Global data flow analysis.
17th Annual Symposium on Foundations of Computer Science, pages 42-56, October 1976.

[FeR69] J. Feldman and P. Rovner.
An ALGOL-based Associative Language.
Communications of the ACM, 12(8):439-449, August 1969.

[Fis81] J. A. Fisher.
Trace Scheduling: A Technique for Global Microcode Compaction.
IEEE Transactions on Computers, 30(7):478-490, July 1981.

[FoK79] M. J. Foster and H. T. Kung.
Design of Special-Purpose VLSI Chips: Example and Opinions.
Technical Report 79-147 , Carnegie-Mellon Univ., September 1979.

[FrS81] E. H. Frank and R. F. Sproull.
Testing and debugging Custom I.C.s.
Computing Surveys, 13(4):425-452, December 1981.

[Gaj82] D. D. Gajski.
The structure of a silicon compiler.
International Conf. on Circuits and Computers, pages 272-276, 1982.

[GFH82] M. Ganapathi, C. N. Fischer and J. L. Hennessey.
Table-driven Code Generation.
ACM Computing Surveys, 14(4):573-592, December 1982.

[Ges72] C. M. Geschke.
Global Program Optimizations.
PhD thesis, Carnegie-Mellon Univ., October 1972.

[Get80] H. Gethoeffer.
SIPROL : A High Level Language for Digital Signal Processing.
International Conf. on Acoustics, Speech and Signal Processing, pages 1056-1059, 1980.

[GlG78] R. S. Glanville and S. L. Graham.
A New Method for Compiler Code Generation.
Conference Record of the Fifth ACM Symposium on Principles of Programming Languages, pages 231-240, January 1978.

[Gre76] C. C. Green.
The design of the PSI Program Synthesis System.
Proceedings Second International Conference on Software Engineering, 1976.

[HaE80] R. M. Haralick and G. L. Elliot.
Increasing Tree Search Efficiency for Constraint Satisfaction Problems.
Artificial Intelligence, 14:263-313, 1980.

[Hec77] M. Hecht.
Flow analysis of Computer Programs.
Elsevier, New York, 1977.

[HSC82] T. S. Hedges, K. H. Slater, G. W. Clow and T. Whitney.
The SICLOPS Silicon Compiler.
International Conf. on Circuits and Computers, 1982.

[JKM68] L. B. Jackson, J. F. Kaiser and H. S. McDonald.
An Approach to the Implementation of Digital Filters.
IEEE Transactions on Audio and Electroacoustics, AU-16(3):413-421, September 1968.

[Joh79] D. Johannsen.
Bristle Blocks: A Silicon Compiler.
16th Design Automation Conf., pages 310-313, 1979.

[Joh] S. C. Johnson.
Code generation for silicon.
Conference Record of the Tenth ACM Symposium on Principles of Programming Languages, pages 14-19.

[Kan82] E. Kant.
Efficiency in Program Synthesis.
UMI Research Press, Ann Arbor, Michigan, 1982.
(also Stanford PhD thesis).

[KaU80] M. Kaplan and J. D. Ullmann.
A Scheme for the Automatic Inference of Variable Types.
Journal of the ACM, 27(1):128-145, January 1980.

[Kar65] B. J. Karafin.
A New Block Diagram Compiler for Simulation of Sampled-Data Systems.
Fall Joint Computer Conference 1965, pages 55-61, AFIPS, 1965.

[KeS82] V. E. Kelley and L. I. Steinberg.
The Critter System: Analyzing Digital Circuits by propagating behaviors and specifications.
Proceedings National Conf. on Artificial Intelligence, pages 284-289, August 1982.

[Ken81] K. Kennedy.
A survey of data flow analysis techniques.
pp. 5-54 in *Program Flow Analysis - Theory and Application*, ed. S. S. Muchnick and N. D. Jones.
PH, 1981.

[Kil73] G. A. Kildall.
A unified approach to global program optimization.
ACM Symposium on Principles of Programming Languages, pages 194-206, 1973.

[Kog81] P. Kogge.
The architecture of pipelined computers.
McGraw Hill, New York, 1981.

[Kop80] G. E. Kopec.
The representation of Discrete-time signals and systems in programs.
PhD thesis, MIT, May 1980.

[Kun81] H. T. Kung.
Why Systolic Architectures.

Technical Report 81-148, Carnegie-Mellon Univ., November 1981.

[LeS81] C. E. Leiserson and J. B. Saxe.
Optimizing Synchronous Systems.
Annual Symposium on Foundations of Computer Science, pages 23-36, 1981.

[LiS] R. J. Lipton and R. Sedgewick.
Lower bounds for VLSI.
Annual Symposium on Foundations of Computer Science ?.

[LAM81] B. H. Liskov, R. Atkinson, E. Moss, J. C. Schaffert, R. Scheifler and A. Snyder.
CLU Reference Manual.
Springer Verlag, Berlin-Heidelberg-New York, 1981.

[Low74] J. R. Low.
Automatic coding: Choice of data structures.
CS-74-452/AIM-242, PhD thesis, Stanford University, August 1974.
(Also published by Birkhauser).

[Low76] B. T. Lowerre.
The HARPY Speech Recognition System.
Technical Report, PhD thesis, Carnegie-Mellon Univ., 1976.

[Lyo80] R. F. Lyon.
Signal processing with VLSI.
in *m.s.*.
Xerox PARC, 1980.

[Lyo81] R. F. Lyon.
A Bit-Serial VLSI Architectural Methodology for Signal Processing.
pp. 131-140 in *VLSI-81*.
Academic Press, New York, NY, 1981.

[Mac77] A. K. Mackworth.
Consistency in Networks of Relations.
Artificial Intelligence, 8:99-118, 1977.

[Mas80] L. M. Masinter.
Global Program Analysis in an Interactive Environment.
SSL-80-1, Xerox PARC, January 1980.
(Also Stanford PhD thesis).

[McS72] D. V. McDermott and G. J. Sussman.
The CONNIVER Reference Manual.
MIT AI Lab. Memo 259, May 1972.

[McD77] D. V. McDermott.
Flexibility and Efficiency a in Computer Program for Designing Circuits.
AI-TR-402, PhD thesis, MIT, June 1977.

[MeC78] C. Mead and L. Conway.
An Introduction to VLSI Systems.
Addison Wesley, Reading, MA, 1978.

[MSS81] T. M. Mitchel, L. Steinberg, R. G. Smith, P. Schooley and V. Kelley.
Representations for reasoning about digital circuits.
LCSR-TR-11, Rutgers University, March 1981.

[MMS79] J. G. Mitchell, W. Maybury and R. Sweet.
Mesa Language Manual.
CSL-79-3, Xerox PARC, April 1979.

[Mol83] D. I. Moldovan.
On the Design of Algorithms for VLSI Systolic Arrays.
Proceedings IEEE, 71(1):113-120, January 1983.

[MuR76] C. T. Mullis and R. A. Roberts.
Roundoff noise in Digital Filters: Frequency transformations and invariants.
IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-24(6):538-550, December 1976.

[New81] A. R. Newton.
Computer aided design of VLSI circuits.
Proceedings of the IEEE, 69(10):1189-1199, October 1981.

[Nil80] N. J. Nilsson.
Principles of Artificial Intelligence.
Tioga Press, Palo Alto, CA, 1980.

[Nud83] B. Nudel.
Consistent-Labeling Problems and their Algorithms: Expected Complexities and Theory-Based Heuristics.
Artificial Intelligence, 21:135-178, 1983.

[Ous81] J. K. Ousterhout.
Caesar: An interactive editor for VLSI layout.
VLSI Design, pages 34-38, Fourth Quarter, 1981.

[PKL80] D. A. Padua, D. J. Kuck and D. H. Lawrie.
High speed multiprocessors and compilation techniques.
IEEE Transactions on Computers, C-29(9):763-776, September 1980.

[Pai83] R. Paige.
Transformational Programming -- Applications to Algorithms and Systems.
Conference Record of the Tenth ACM Symposium on Principles of Programming Languages, pages 73-87, 1983.

[PTS79] A. Parker, D. Thomas, D. Sieworek, M. Barbacci, L. Hafer, G. Leive and J. Kim.
The Carnegie-Mellon Univ. Design Automation System.
16th Design Automation Conf., pages 73-80, 1979.

[PaW81] A. C. Parker and W. T. Wilner.
Microprogramming - The challenges of VLSI.
National Computer Conference, pages 63-68, AFIPS, 1981.

[PDS77] D. Persky, D. N. Deutsch and D. G. Schweikert.
LTX -- A Minicomputer-Based System For Automated LSI Layout.
Journal of Design Automation and Fault-Tolerant Computing, 1(3):217-255, May 1977.

[RaL77] C. V. Ramamoorthy and H. F. Li.
Pipelined Architecture.
ACM Computing Surveys, 9(1):61-102, March 1977.

[Ram80] R. J. Ramirez.
Efficient algorithms for selecting efficient data storage structures.
CS-80-18, PhD thesis, University of Waterloo, March 1980.

[Ram79] L. H. Ramshaw.
Formalizing the Analysis of Algorithms.
CSL-79-5, Xerox PARC, June 1979.
(also STAN-CS-79-741, Stanford University).

[RiW80] M. Richards and C. Whitby-Streevens.
BCPL - The language and its compiler.
Cambridge University Press, 1980.

[Riv82] R. Rivest.
The PI (Placement and Interconnect) System.
19th Design Automation Conf., March 1982.
(Also MIT memo 82-74).

[Ros77] B. K. Rosen.
High Level Data-flow analysis.
Communications of the ACM, 20(10):712-724, October 1977.

[Rov] P. D. Rovner.
Automatic Representation Selection for Associative Data Structures.
Univ. of Rochester TR-10, PhD thesis, Harvard University.

[RoT] L. A. Rowe and F. M. Tonge.
Algorithms for the synthesis of implementation structures.
Univ. of California, Irvine TR-91.

[Sac75] E. D. Sacerdoti.
A structure for plans and behavior.
Technical Note 109, SRI, August 1975.

[San78] E. Sandewall.
Programming in an Interactive Environment: The "Lisp" experience.
ACM Computing Surveys, 10(1):35-71, March 1978.

[Sau79] S. E. Saunders.
Compiling Customized Executable Representations and Interpreters.
Technical Report 79-127, PhD thesis, Carnegie-Mellon Univ., June 1979.

[SSS81] E. Schonberg, J. T. Schwartz and M. Sharir.
An automatic Technique for selection of data representations in SETL programs.
ACM Transactions on Programming Languages and Systems, 3:126-143, April 1981.

[Sch5)] J. T. Schwartz.
On Programming: An Interim Report on the SETL Project.
Courant Institute, New York University, 1973 (second ed. 1975).

[Sch75] J. T. Schwartz.
Automatic Data Structure Choice in a Language of Very High Level.
Communications of the ACM, pages 722-728, December 1975.

[Shr82] H. E. Shrobe.
The Data Path Generator.
Proceedings of MIT VLSI Conference, pages 175-181, 1982.

[SSC82] J. M. Siskind, J. R. Southard and K. W. Crouch.
Generating custom high performance VLSI designs from succinct algorithmic descriptions.
Proceedings of MIT VLSI Conference, pages 28-39, 1982.

[Sny82] L. Snyder.
Recognition and Selection of Idioms for Code Optimization.
Acta Informatica, 17:327-348, 1982.

[Ste80a] G. L. Steele.
The Definition and Implementation of a Computer Programming Language based on Constraints.

AI-TR-595, PhD thesis, MIT, August 1980.

[Ste80b] M. J. Stefik.
Planning with Constraints.
Technical Report 80-794, PhD Thesis, Stanford University, January 1980.

[Ste81a] M. J. Stefik.
Planning and Meta-Planning (MOLGEN: Part 2).
Artificial Intelligence, 16:141-170, 1981.

[Ste81b] M. J. Stefik.
Planning with Constraints (MOLGEN: Part 1).
Artificial Intelligence, 16:111-140, 1981.

[Str78] W. D. Strecker.
VAX 11/780 - A Virtual Address extension to the December PDP-11 family.
National Computer Conference, pages 967-980, AFIPS, 1978.

[Sus77] G. J. Sussman.
Electrical Design - A problem for Artificial Intelligence Research.
Fifth International Joint Conf. on Artificial Intelligence, 2:894-900, August 1977 .

[SuS75] G. J. Sussman and R. M. Stallman.
Heuristic Techniques in Computer-Aided Circuit Analysis.
IEEE Transactions on Circuits and Systems, CAS-22(11):857-865, November 1975.

[Sus75] G. J. Sussman.
A Computer Model of Skill Acquisition.
Elsevier, New York, 1975.

[Sui77] N. Suzuki and K. Ishihata.
Implementation of an Array Bound checker.
Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, pages 132-143, January 1977.
(also Carnegie-Mellon Univ. Technical Report)

[Ten74] A. M. Tenenbaum.
Type Determination for Very High Level Languages.
PhD thesis, Courant Institute, October 1974.

[Tho80] C. D. Thompson.
A Complexity theory for VLSI.
Technical Report 80-140, PhD thesis, Carnegie-Mellon Univ., August 1980.

[ToR80] F. W. Tompa and R. J. Ramirez.
An aid for the Selection of Efficient Storage Structures.
CS-80-46, University of Waterloo, October 1980.

[ToW77] H. C. Torng and N. C. Wilhelm.
The Optimal Interconnection of Circuit Modules in Microprocessor and Digital System Design.
IEEE Transactions on Computers, C-26(5):450-457, May 1977.

[Tre82] P. C. Treleaven.
VLSI Processor architectures.
Computer, 15(6):33-45, June 1982.

[TBH82] P. C. Treleaven, D. R. Brownbridge and R. P. Hopkins.
Data-Driven and Demand-Driven Computer Architecture.
Computing Surveys, 14(1):93-144, March 1982.

[TsS81] C. Tseng and D. P. Sieworek.
The Modeling and Synthesis of Bus Systems.
18th Design Automation Conf., pages 471-478, 1981.

[Wal75] D. Waltz.
Using constraints in computer scene understanding.
pp. 19-92 in *Psychology of Computer Vision*, ed. P. H. Winston.
McGraw-Hill, New York, 1975.

[Wan82] M. Wand.
Semantics-Directed Machine Architecture.
Conference Record of the Ninth ACM Symposium on Principles of Programming Languages, pages 234-241, January 1982.

[Weg75] B. Wegbreit.
Property extraction on well-founded property sets.
IEEE Transactions on Software Engineering, SE-1(3):270-285, September 1975.

[WiP83] T. W. Williams and K. P. Parker.
Design for testability - a survey.
Proceedings of the IEEE, 71(1):98-112, January 1983.

[Zim81] G. Zimmerman.
VLSI design with the MIMOLA design system.
IEE Conf. Publication No. 200, pages 277-280, 1981.

[Zim79] G. Zimmermann.
The MIMOLA Design System - A Computer aided digital Processor Design Method.
16th Design Automation Conf., pages 53-63, 1979.

[Zim80] G. Zimmermann.
MDS - The MIMOLA Design Method.
Journal of Digital Systems, 4(3):337-369, 1980.

Citation Index

Index

END
FILMED
4-86
DTIC